

# UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

---

*Corso di Laurea Magistrale in Ingegneria Informatica*

---



## ELABORATO DI ARCHITETTURA DEI SISTEMI DIGITALI

*Prof.ssa Alessandra De Benedictis*

a.a. 2023-24

Studenti:

Federica Del Vecchio, M63001587

Esposito Claudia Antonella, M63001625

Tommaso Di Lillo, M63001642

Carolina Di Donato, M63001591

## Sommario

Capitolo 1: Reti Combinatorie Elementari .....	4
Esercizio 1: Multiplexer 16:1.....	4
Progetto e architettura .....	4
Implementazione.....	6
Simulazione.....	10
Sintesi su board di sviluppo .....	13
Esercizio 2: Sistema ROM+M .....	17
Progetto e architettura .....	17
Implementazione.....	17
Simulazione.....	19
Sintesi su board di sviluppo .....	23
Capitolo 2: Reti Sequenziali Elementari.....	24
Esercizio 3: Riconoscitore di sequenze .....	24
Progetto e architettura .....	24
Implementazione.....	24
Simulazione.....	27
Sintesi su board di sviluppo .....	29
Esercizio 4: Shift Register .....	32
Progetto e architettura .....	32
Implementazione.....	33
Simulazione.....	38
Esercizio 5: Cronometro .....	43
Progetto e architettura .....	43
Implementazione.....	43
Simulazione.....	46
Sintesi su board di sviluppo .....	48
Esercizio 6: Sistema di lettura-elaborazione-scrittura PO_PC .....	59
Progetto e architettura .....	59
Implementazione.....	60
Simulazione.....	66
Sintesi su board di sviluppo .....	68
Timing analysis.....	70
Capitolo 3: Macchine Aritmetiche .....	73
Esercizio 7: Moltiplicatore di Booth.....	73
Progetto e architettura .....	73

Implementazione .....	77
Simulazione.....	86
Sintesi su board di sviluppo .....	88
Capitolo 4: Comunicazione con Handshaking .....	94
Esercizio 8: Comunicazione con Handshaking.....	94
Progetto e architettura .....	94
Implementazione.....	97
Simulazione.....	108
Capitolo 5: Processore .....	110
Esercizio 9: Processore .....	110
Capitolo 6: Interfaccia Seriale.....	120
Esercizio 10: RS232 .....	120
Progetto e architettura .....	120
Implementazione.....	124
Simulazione.....	134
Capitolo 7: Switch Multistadio.....	136
Esercizio 11: Switch Multistadio .....	136
Progetto e architettura .....	136
Implementazione.....	138
Simulazione.....	143

# Capitolo 1: Reti Combinatorie Elementari

## Esercizio 1: Multiplexer 16:1

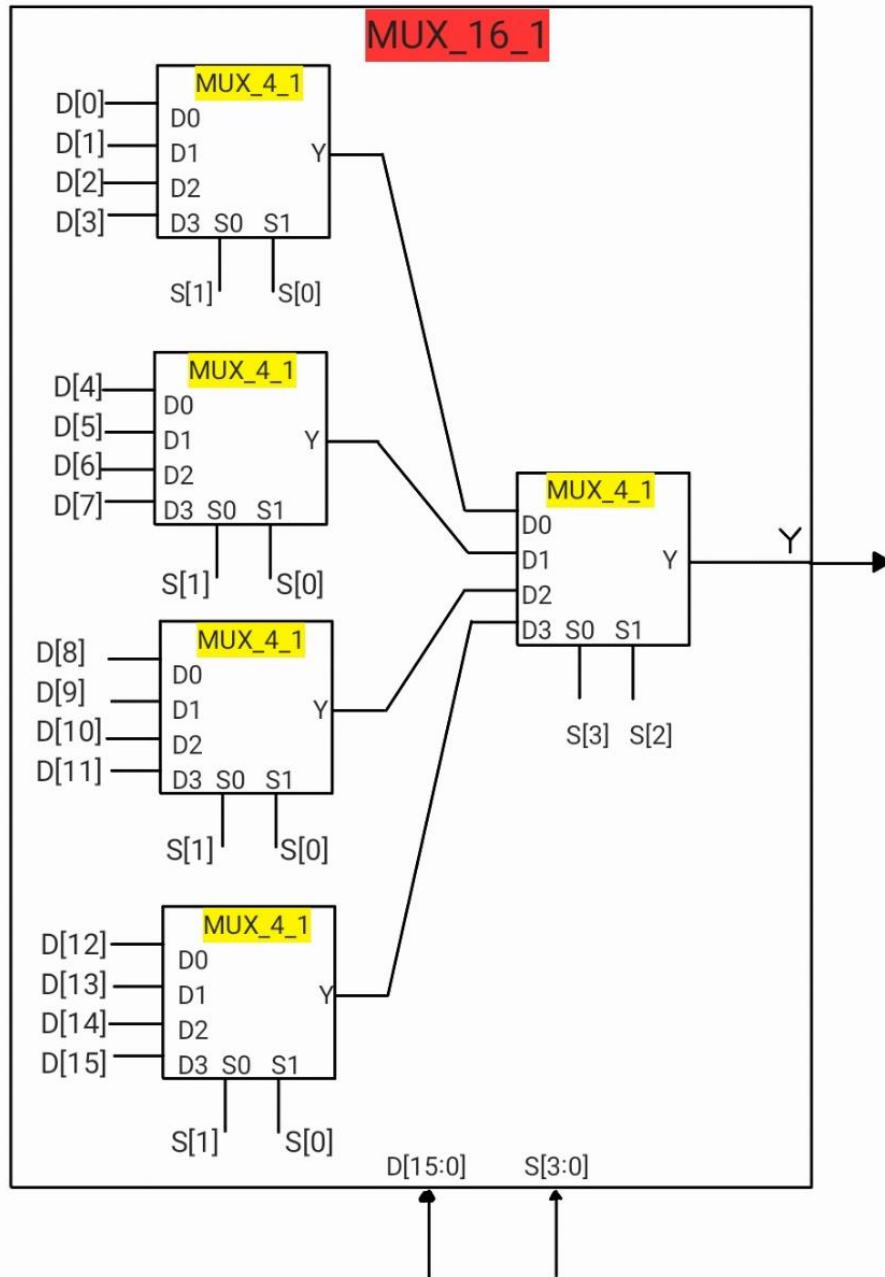
### *Progetto e architettura*

Per implementare il multiplexer indirizzabile 16:1, si è adottato un approccio di progettazione per composizione, sfruttando un multiplexer 4:1 come elemento base.

La realizzazione del multiplexer 4:1 ha coinvolto la definizione comportamentale di un modulo (**mux\_4\_1**) con sei ingressi (**D0, D1, D2, D3, S0, S1**) e un'uscita (**Y**). La logica di selezione è gestita in un processo comportamentale e la scelta viene fatta in base a due segnali di selezione in input (**S0, S1**). Ad esempio, se sia S0 che S1 sono alti, l'uscita Y sarà collegata a D3.

L'entity **mux\_16\_1** presenta, invece, due ingressi (**D, S**) e una singola uscita (**Y**). In particolare, D è un vettore di 16 bit che rappresenta le linee di dati in ingresso, mentre S è un vettore di 4 bit che funge da selezionatore. L'architettura del multiplexer 16:1 è implementata in modo strutturale: si impiegano cinque istanze del multiplexer 4:1 come “building block” per la costruzione del multiplexer indirizzabile 16:1.

Nell'immagine sottostante sono illustrati i collegamenti tra le uscite e gli ingressi del multiplexer 16:1 e le uscite e gli ingressi dei multiplexer 4:1. Questa rappresentazione offre una chiara visione dell'organizzazione gerarchica ad albero che si sviluppa da tali connessioni.

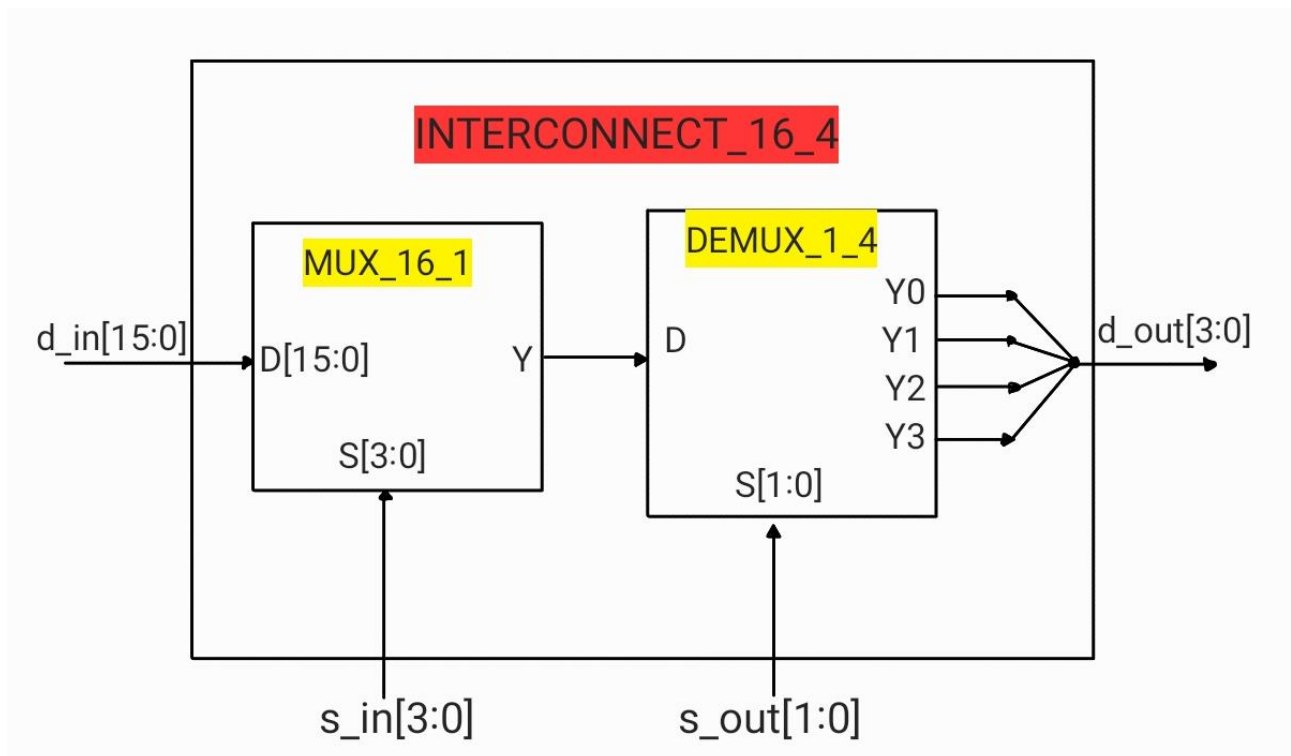


Per implementare una rete di interconnessione a 16 sorgenti e 4 destinazioni (**interconnect\_16\_4**), si è adottato un approccio di progettazione strutturale. Il multiplexer indirizzabile 16:1 precedentemente sviluppato è un componente base che viene combinato in serie con un demultiplexer 1:4 (**demux\_1\_4**), sempre realizzato in modo “Behavioral”.

Il modulo **mux\_4\_to\_1** rappresenta un multiplexer con quattro ingressi (**D0**, **D1**, **D2**, **D3**) e un'uscita (**Y**). I segnali di selezione **S0** e **S1** controllano quale ingresso viene instradato all'uscita; ad esempio, se **S0** e **S1** sono entrambi bassi, l'output sarà collegata a **D0**.

Nel network, il multiplexer 16:1 è incaricato di selezionare l'input da instradare in base a linee di selezione della sorgente (**s\_in**). Contemporaneamente, il demultiplexer 1:4 indirizza il segnale prescelto verso la destinazione specificata attraverso segnali di selezione dedicati (**s\_out**).

Di seguito è riportata l'immagine dell'architettura del sistema di interconnessione completo.



### Implementazione

Sono successivamente riportate le descrizioni VHDL del multiplexer 4:1, del multiplexer 16:1, del demultiplexer 4:1 e della rete di interconnessione 16:4.

#### **mux\_4\_to\_1.vhd**

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mux_4_to_1 is
    Port ( D0, D1, D2, D3 : in STD_LOGIC;
          S0, S1 : in STD_LOGIC;
          Y : out STD_LOGIC);
end mux_4_to_1;

architecture Behavioral of mux_4_to_1 is
begin
    process (S0, S1, D0, D1, D2, D3)
    begin
        if (S0 = '0' and S1 = '0') then
            Y <= D0;
        elsif (S0 = '0' and S1 = '1') then
            Y <= D1;
        
```

```

        elsif (S0 = '1' and S1 = '0') then
            Y <= D2;
        elsif (S0 = '1' and S1 = '1') then
            Y <= D3;
        else
            Y <= '-';
        end if;
    end process;
end Behavioral;

```

---

## **mux\_16\_1.vhd**

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mux_16_1 is
    Port ( D : in STD_LOGIC_VECTOR(15 downto 0);
          S : in STD_LOGIC_VECTOR(3 downto 0);
          Y : out STD_LOGIC);
end mux_16_1;

architecture Structural of mux_16_1 is
    signal intermediate_signals : STD_LOGIC_VECTOR(3 downto 0);
    signal Y_temp : STD_LOGIC;

    component mux_4_to_1
        Port ( D0, D1, D2, D3 : in STD_LOGIC;
              S0, S1 : in STD_LOGIC;
              Y : out STD_LOGIC);
    end component;

begin
    MUX0: mux_4_to_1 port map (
        D0 => D(0), D1 => D(1), D2 => D(2), D3 => D(3),
        S0 => S(1), S1 => S(0),
        Y => intermediate_signals(0)
    );

    MUX1: mux_4_to_1 port map (
        D0 => D(4), D1 => D(5), D2 => D(6), D3 => D(7),
        S0 => S(1), S1 => S(0),
        Y => intermediate_signals(1)
    );

    MUX2: mux_4_to_1 port map (
        D0 => D(8), D1 => D(9), D2 => D(10), D3 => D(11),
        S0 => S(1), S1 => S(0),

```

```

        Y => intermediate_signals(2)
    );

    MUX3: mux_4_to_1 port map (
        D0 => D(12), D1 => D(13), D2 => D(14), D3 => D(15),
        S0 => S(1), S1 => S(0),
        Y => intermediate_signals(3)
    );

    MUX_final: mux_4_to_1 port map (
        D0 => intermediate_signals(0),
        D1 => intermediate_signals(1),
        D2 => intermediate_signals(2),
        D3 => intermediate_signals(3),
        S0 => S(3),
        S1 => S(2),
        Y => Y_temp
    );

    Y <= Y_temp;
end Structural;

```

---

#### demux\_1\_4.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity demux_1_4 is
    Port ( D : in STD_LOGIC;
          S : in STD_LOGIC_VECTOR(1 downto 0);
          Y0, Y1, Y2, Y3 : out STD_LOGIC);
end demux_1_4;

architecture Behavioral of demux_1_4 is
begin
    process (D, S)
    begin
        case S is
            when "00" =>
                Y0 <= D;
                Y1 <= '0';
                Y2 <= '0';
                Y3 <= '0';
            when "01" =>
                Y0 <= '0';
                Y1 <= D;
                Y2 <= '0';
        end case;
    end process;
end Behavioral;

```



```

        Y3 <= '0';
    when "10" =>
        Y0 <= '0';
        Y1 <= '0';
        Y2 <= D;
        Y3 <= '0';
    when "11" =>
        Y0 <= '0';
        Y1 <= '0';
        Y2 <= '0';
        Y3 <= D;
    when others =>
        Y0 <= '-';
        Y1 <= '-';
        Y2 <= '-';
        Y3 <= '-';
    end case;
end process;
end Behavioral;

```

---

## interconnect\_16\_4.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity interconnect_16_4 is
    Port (
        d_in  : in  STD_LOGIC_VECTOR(15 downto 0);
        s_in  : in  STD_LOGIC_VECTOR(3  downto 0);
        s_out : in  STD_LOGIC_VECTOR(1  downto 0);
        d_out : out STD_LOGIC_VECTOR(3  downto 0)
    );
end interconnect_16_4;

architecture Structural of interconnect_16_4 is
    component mux_16_1
        port (
            D : in  STD_LOGIC_VECTOR(15 downto 0);
            S : in  STD_LOGIC_VECTOR(3  downto 0);
            Y : out STD_LOGIC
        );
    end component;

    component demux_1_4
        port (
            D : in  STD_LOGIC;
            S : in  STD_LOGIC_VECTOR(1 downto 0);
            Y0 : out STD_LOGIC;
            Y1 : out STD_LOGIC;

```

```

        Y2 : out STD_LOGIC;
        Y3 : out STD_LOGIC
    );
end component;

signal mux_out : STD_LOGIC;

begin
    mux_16_inst : mux_16_1
        port map (
            D => d_in,
            S => s_in,
            Y => mux_out
        );

    demux_4_inst : demux_1_4
        port map (
            D  => mux_out,
            S  => s_out,
            Y0 => d_out(0),
            Y1 => d_out(1),
            Y2 => d_out(2),
            Y3 => d_out(3)
        );
end Structural;

```

---

### Simulazione

Sono stati sviluppati testbench per verificare il corretto funzionamento di ciascun modulo, tuttavia, ci si concentra su due in particolare.

Il testbench per il multiplexer 16:1 (**mux\_16\_1\_tb**) è stato progettato per simulare il multiplexer indirizzabile in diverse condizioni di input (**D\_input**) e selezione(**S\_input**). Lo scopo è verificare la corrispondenza tra l'output risultante (**Y\_output**) e quello atteso dato dalla specifica situazione in esame.

---

### mux\_16\_1\_tb.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux_16_1_tb is
end mux_16_1_tb;

architecture testbench of mux_16_1_tb is
    signal D_input : STD_LOGIC_VECTOR(15 downto 0) := "0000000000000000";
    signal S_input : STD_LOGIC_VECTOR(3  downto 0) := "0000";
    signal Y_output : STD_LOGIC;

    component mux_16_1
        Port ( D : in STD_LOGIC_VECTOR(15 downto 0);

```

```

        S : in STD_LOGIC_VECTOR(3 downto 0);
        Y : out STD_LOGIC);
end component;

begin
    MUX16to1 : mux_16_1
        port map (D => D_input, S => S_input, Y => Y_output);

    stimulus : process
    begin
        D_input <= "0000000000000001";
        S_input <= "0000";
        wait for 10 ns;

        D_input <= "1000000000000100";
        S_input <= "1110";
        wait for 10 ns;

        D_input <= "0000000000001000";
        S_input <= "0011";
        wait for 10 ns;

        D_input <= "0000000000010000";
        S_input <= "0110";
        wait for 10 ns;

        D_input <= "1000000100000000";
        S_input <= "1111";
        wait for 10 ns;

        D_input <= "0000000000000001";
        S_input <= "0000";
        wait for 10 ns;

        D_input <= "0000000000000100";
        S_input <= "0010";
        wait for 10 ns;

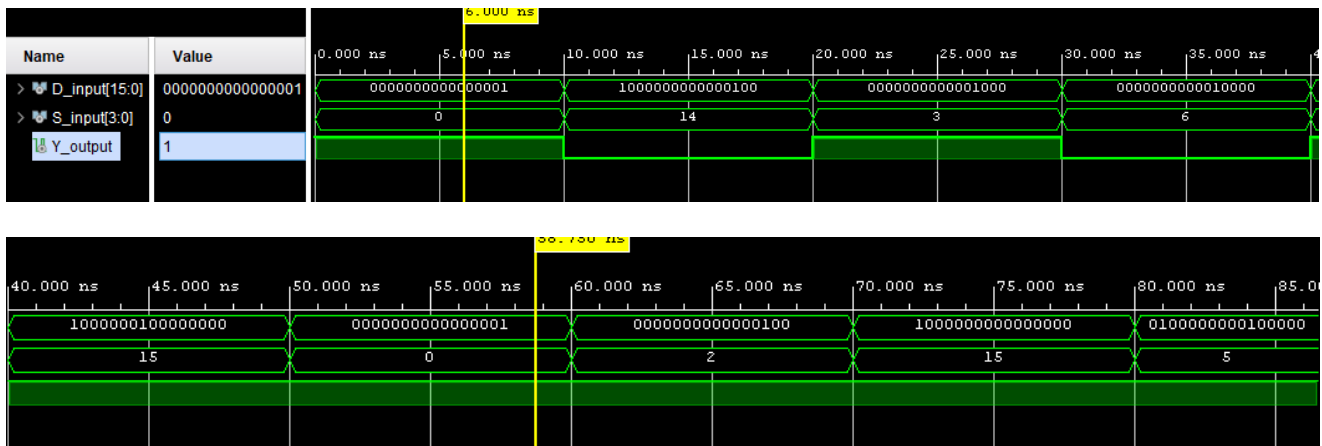
        D_input <= "1000000000000000";
        S_input <= "1111";
        wait for 10 ns;

        D_input <= "0100000000100000";
        S_input <= "0101";
        wait for 10 ns;

        wait;
    end process stimulus;

end testbench;

```



Il testbench **interconnect\_16\_4\_tb** verifica il funzionamento dell'intera rete di interconnessione a 16 sorgenti e 4 uscite. Applicando sequenze di ingressi a 16 bit (**d\_in\_tb**) e segnali di selezione (**s\_in\_tb**, **s\_out\_tb**), il testbench valuta l'adeguato instradamento del segnale attraverso la rete, analizzando l'uscita ottenuta (**d\_out\_tb**).

#### interconnect\_16\_4\_tb.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity interconnect_16_4_tb is
end interconnect_16_4_tb;

architecture testbench of interconnect_16_4_tb is
    signal d_in_tb    : STD_LOGIC_VECTOR(15 downto 0);
    signal s_in_tb    : STD_LOGIC_VECTOR(3  downto 0);
    signal s_out_tb   : STD_LOGIC_VECTOR(1  downto 0);
    signal d_out_tb   : STD_LOGIC_VECTOR(3  downto 0);

    component interconnect_16_4
        port (
            d_in  : in  STD_LOGIC_VECTOR(15 downto 0);
            s_in  : in  STD_LOGIC_VECTOR(3  downto 0);
            s_out : in  STD_LOGIC_VECTOR(1  downto 0);
            d_out : out STD_LOGIC_VECTOR(3  downto 0)
        );
    end component;

begin
    uut : interconnect_16_4
        port map (
            d_in  => d_in_tb,
            s_in  => s_in_tb,
            s_out => s_out_tb,
```

```

        d_out => d_out_tb
    );

stimuli : process
begin
    d_in_tb  <= (others => '0');
    s_in_tb  <= "0000";
    s_out_tb <= "00";

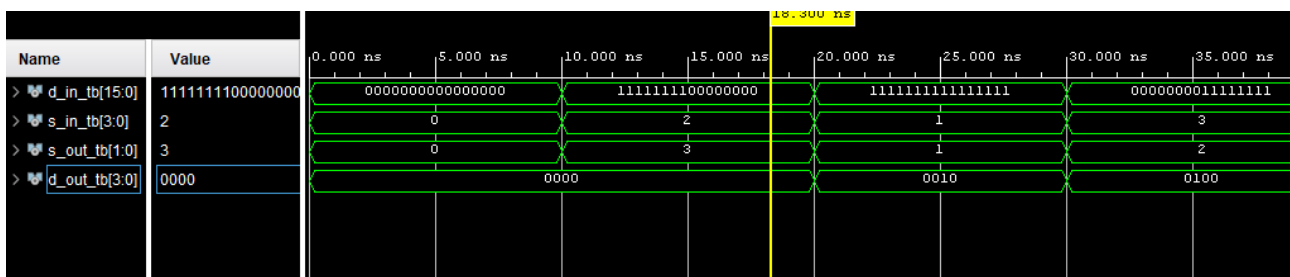
    wait for 10 ns;
    d_in_tb  <= "1111111100000000";
    s_in_tb  <= "0010";
    s_out_tb <= "11";
    wait for 10 ns;

    d_in_tb  <= "1111111111111111";
    s_in_tb  <= "0001";
    s_out_tb <= "01";
    wait for 10 ns;

    d_in_tb  <= "0000000011111111";
    s_in_tb  <= "0011";
    s_out_tb <= "10";
    wait for 10 ns;

    wait;
end process;
end testbench;

```



### Sintesi su board di sviluppo

La sintesi sul board Nexys A7-100T richiede la scrittura di un modulo dedicato. L'entity **interconnect\_16\_4\_board** presenta diverse porte di input e output necessarie per interfacciarsi con il board di sviluppo:

- **value14\_in** è un vettore di 14 segnali collegati ai primi 14 switch, da **U12** a **J15**, i quali vengono utilizzati per inserire i dati in ingresso e la sorgente e la destinazione.
- **load\_sel** è il segnale collegato al pulsante **P18**; quando esso viene premuto si carica l'informazione della destinazione dagli switch **U12** e **H6** e quella della sorgente dagli switch **T13** a **T8**.

- **load\_first\_part** è il segnale collegato al bottone **P17**; quando esso viene premuto si carica la prima metà dei dati in ingresso, quella meno significativa, a partire dagli switch **R13** a **J15**.
- **load\_second\_part** è il segnale collegato al pulsante **M17** il quale quando viene premuto carica la seconda metà dei dati, quelli più significativi, sempre dagli switch **R13** a **J15**.
- **reset** è collegato al button **N17** e quando questo viene premuto tutte i segnali interni a **interconnect\_16\_4\_board** che contengono informazioni precedentemente inserite sono azzerati.
- **led** è un vettore di output a 4 bit collegato ai led da **N14** a **H17**, utilizzati per visualizzare l'output del sistema.

## interconnect\_16\_4\_board.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity interconnect_16_4_board is
    Port (
        load_first_part : in  STD_LOGIC;
        load_second_part : in  STD_LOGIC;
        load_sel : in  STD_LOGIC;
        reset : in  STD_LOGIC;
        value14_in : in  STD_LOGIC_VECTOR(13 downto 0);
        led : out  STD_LOGIC_VECTOR(3 downto 0)
    );
end interconnect_16_4_board;

architecture Structural of interconnect_16_4_board is
    component interconnect_16_4
        port (
            d_in  : in  STD_LOGIC_VECTOR(15 downto 0);
            s_in  : in  STD_LOGIC_VECTOR(3 downto 0);
            s_out : in  STD_LOGIC_VECTOR(1 downto 0);
            d_out : out STD_LOGIC_VECTOR(3 downto 0)
        );
    end component;

    signal inter_out : STD_LOGIC_VECTOR(3 downto 0) := (others => '0');
    signal inter_in  : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
    signal sel_in    : STD_LOGIC_VECTOR(3 downto 0) := (others => '0');
    signal sel_out   : STD_LOGIC_VECTOR(1 downto 0) := (others => '0');

begin
    intconn_16_4 : interconnect_16_4
        port map (
            d_in  => inter_in,
            s_in  => sel_in,
            s_out => sel_out,
            d_out => inter_out
        );
end;
```

```

main : process (reset, load_sel, load_first_part, load_second_part)
begin
    if reset = '1' then
        inter_in <= (others => '0');
    elsif load_sel = '1' then
        sel_in <= value14_in(11 downto 8);
        sel_out <= value14_in(13 downto 12);
    elsif load_first_part = '1' then
        inter_in(7 downto 0) <= value14_in(7 downto 0);
    elsif load_second_part = '1' then
        inter_in(15 downto 8) <= value14_in(7 downto 0);
    end if;
END PROCESS;

led <= inter_out;

end Structural;

```

---

Dopo sono riportate le linee che è stato necessario decommentare e opportunamente modificare nel file dei vincoli "Nexys-A7-100T-Master.xdc" per poter implementare quanto appena descritto.

---

### ##Switches

```

set_property -dict { PACKAGE_PIN J15  IOSTANDARD LVCMOS33 } [get_ports { value14_in[0]
}]; #IO_L24N_T3_RS0_15 Sch=sw[0]

set_property -dict { PACKAGE_PIN L16  IOSTANDARD LVCMOS33 } [get_ports { value14_in[1]
}]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]

set_property -dict { PACKAGE_PIN M13  IOSTANDARD LVCMOS33 } [get_ports {
value14_in[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]

set_property -dict { PACKAGE_PIN R15  IOSTANDARD LVCMOS33 } [get_ports { value14_in[3]
}]; #IO_L13N_T2_MRCC_14 Sch=sw[3]

set_property -dict { PACKAGE_PIN R17  IOSTANDARD LVCMOS33 } [get_ports { value14_in[4]
}]; #IO_L12N_T1_MRCC_14 Sch=sw[4]

set_property -dict { PACKAGE_PIN T18  IOSTANDARD LVCMOS33 } [get_ports { value14_in[5]
}]; #IO_L7N_T1_D10_14 Sch=sw[5]

set_property -dict { PACKAGE_PIN U18  IOSTANDARD LVCMOS33 } [get_ports { value14_in[6]
}]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]

set_property -dict { PACKAGE_PIN R13  IOSTANDARD LVCMOS33 } [get_ports { value14_in[7]
}]; #IO_L5N_T0_D07_14 Sch=sw[7]

```

```
set_property -dict { PACKAGE_PIN T8   IOSTANDARD LVCMOS18 } [get_ports { value14_in[8]
}]; #IO_L24N_T3_34 Sch=sw[8]
```

```
set_property -dict { PACKAGE_PIN U8   IOSTANDARD LVCMOS18 } [get_ports { value14_in[9]
}]; #IO_L25_34 Sch=sw[9]
```

```
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports {
value14_in[10] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
```

```
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports {
value14_in[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
```

```
set_property -dict { PACKAGE_PIN H6     IOSTANDARD LVCMOS33 } [get_ports {
value14_in[12] }]; #IO_L24P_T3_35 Sch=sw[12]
```

```
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports {
value14_in[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
```

## LEDs

```
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { led[0] }];
#IO_L18P_T2_A24_15 Sch=led[0]
```

```
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { led[1] }];
#IO_L24P_T3_RS1_15 Sch=led[1]
```

```
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 } [get_ports { led[2] }];
#IO_L17N_T2_A25_15 Sch=led[2]
```

```
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 } [get_ports { led[3] }];
#IO_L8P_T1_D11_14 Sch=led[3]
```

##Buttons

```
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { reset }];
#IO_L9P_T1_DQS_14 Sch=btnc
```

```
set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports {
load_first_part }]; #IO_L12P_T1_MRCC_14 Sch=btnl
```

```
set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports {
load_second_part }]; #IO_L10N_T1_D15_14 Sch=btnr
```

```
set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { load_sel
}]; #IO_L9N_T1_DQS_D13_14 Sch=btnd
```



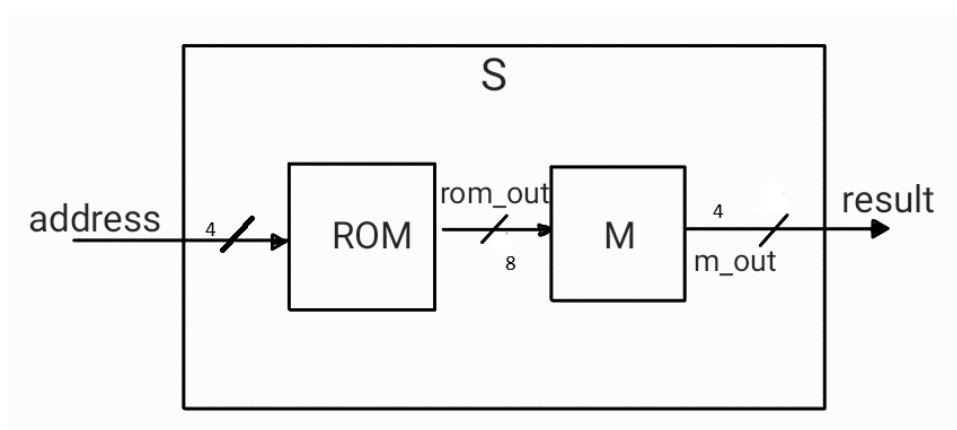
## Esercizio 2: Sistema ROM+M

### Progetto e architettura

Il sistema totale S è realizzato per composizione di due componenti: una memoria di sola lettura di tipo ROM che contiene 16 parole a 8 bit e una macchina di tipo M che trasforma word a 8 bit in word a 4 bit secondo una specifica funzione.

Il sistema S ha un ingresso **address** che è un vettore di 4 bit e un'uscita **result** che è anch'essa un vettore di 4 bit. Tale sistema utilizza due segnali intermedi: il primo è **rom\_out** che è un vettore di 8 bit che rappresenta l'uscita della memoria ROM, mentre il secondo è **m\_out** che è un vettore di 4 bit.

Il vettore **address** va in input alla componente ROM che dà in output il vettore di 8 bit letto in questa posizione (**d\_out**), il quale è poi collegato all'ingresso **x** della macchina M. Essa è una macchina combinatoria elementare che prende i 4 bit meno significativi di **x**, li nega e li assegna all'uscita **y** da 4 bit. Il segnale **m\_out** connette l'output del blocco M con l'output dell'intero sistema S.



### Implementazione

L'entity ROM è realizzata attraverso un'architettura comportamentale; invece, la macchina M è progettata da un livello di astrazione dataflow. S presenta chiaramente un'architettura "Structural".

---

### S.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity S is port(
    address : in  std_logic_vector(3 downto 0);
    result  : out std_logic_vector(3 downto 0)
);
end entity S;

architecture S_a of S is
    signal rom_out  : std_logic_vector(7 downto 0);
    signal m_out    : std_logic_vector(3 downto 0);

begin
```

```

memory_rom : entity work.ROM port map(
    address => address,
    d_out => rom_out
);

machine_m : entity work.M port map(
    x => rom_out,
    y => m_out
);

result <= m_out;
end architecture S_a;

```

---

## ROM.vhd

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ROM is port(
    address : in std_logic_vector(3 downto 0);
    d_out    : out std_logic_vector(7 downto 0)
);
end entity ROM;

architecture RTL of ROM is
    type MEMORY_16_8 is array (0 to 15) of std_logic_vector(7 downto 0);

    constant ROM_16_8 : MEMORY_16_8 := (
        "00000000",
        "00000001",
        "00000010",
        "00000011",
        "00000100",
        "00000101",
        "00000110",
        "00000111",
        "00001000",
        "00001001",
        "00001010",
        "00001011",
        "00001100",
        "00001101",
        "00001110",
        "00001111"
    );

begin
    main : process(address)

```

```
begin
    d_out <= ROM_16_8(to_integer(unsigned(address)));
end process main;
end architecture RTL;
```

---

## M.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity M is port(
    x : in  std_logic_vector(7 downto 0);
    y : out std_logic_vector(3 downto 0)
);
end entity M;

architecture Dataflow of M is
begin
    process(x)
    begin
        y(3 downto 0) <= not(x(3 downto 0));
    end process;
end architecture Dataflow;
```

---

## Simulazione

---

## M\_tb.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity M_tb is
end M_tb;

architecture testbench of M_tb is
    signal x_tb : STD_LOGIC_VECTOR(7 downto 0);
    signal y_tb : STD_LOGIC_VECTOR(3 downto 0);

    component M
        port (
            x : in  STD_LOGIC_VECTOR(7 downto 0);
            y : out STD_LOGIC_VECTOR(3 downto 0)
        );
    end component;
```

```

begin
    uut : M
        port map (
            x => x_tb,
            y => y_tb
        );

    stimulus : process
    begin
        wait for 10 ns;

        x_tb <= "00000000";
        wait for 10 ns;

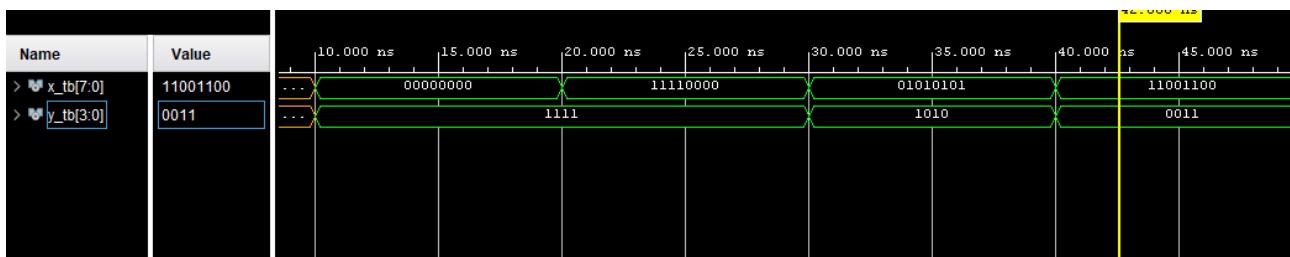
        x_tb <= "11110000";
        wait for 10 ns;

        x_tb <= "01010101";
        wait for 10 ns;

        x_tb <= "11001100";
        wait for 10 ns;

        wait;
    end process stimulus;
end testbench;

```



## ROM\_tb.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ROM_tb is
end ROM_tb;

architecture testbench of ROM_tb is
    signal address_tb : STD_LOGIC_VECTOR(3 downto 0);
    signal d_out_tb   : STD_LOGIC_VECTOR(7 downto 0);

    component ROM

```

```

        port (
            address : in  STD_LOGIC_VECTOR(3 downto 0);
            d_out    : out STD_LOGIC_VECTOR(7 downto 0)
        );
    end component;

begin
    uut : ROM
        port map (
            address => address_tb,
            d_out    => d_out_tb
        );

    stimulus : process
    begin
        wait for 10 ns;

        address_tb <= "0000";
        wait for 10 ns;

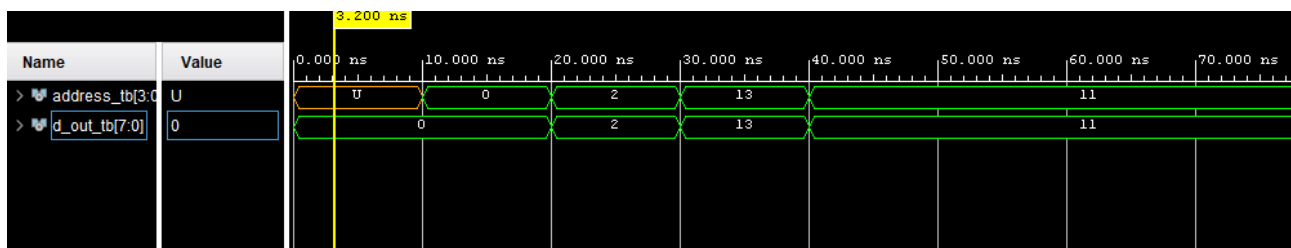
        address_tb <= "0010";
        wait for 10 ns;

        address_tb <= "1101";
        wait for 10 ns;

        address_tb <= "1011";
        wait for 10 ns;

        wait;
    end process stimulus;
end testbench;

```



## S\_tb.vhd

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity S_tb is
end entity S_tb;

```

```

architecture testbench of S_tb is
    signal address : std_logic_vector(3 downto 0) := "0000";
    signal result : std_logic_vector(3 downto 0);

    component S port(
        address : in  std_logic_vector(3 downto 0);
        result : out std_logic_vector(3 downto 0)
    );
    end component S;

begin
    uut: S port map(
        address => address,
        result  => result
    );

    stimuli : process
    begin
        address <= "0000";
        wait for 10 ns;

        address <= "0001";
        wait for 10 ns;

        address <= "1000";
        wait for 10 ns;

        address <= "1101";
        wait for 10 ns;

        address <= "1111";
        wait for 10 ns;

        wait;
    end process;

    process
    begin
        wait for 50 ns;

        assert result = "0000" report "Errore all'indirizzo 0" severity
error;
        assert result = "0001" report "Errore all'indirizzo 1" severity
error;
        assert result = "1000" report "Errore all'indirizzo 8" severity
error;
        assert result = "1101" report "Errore all'indirizzo 13" severity
error;
    end process;
end testbench;

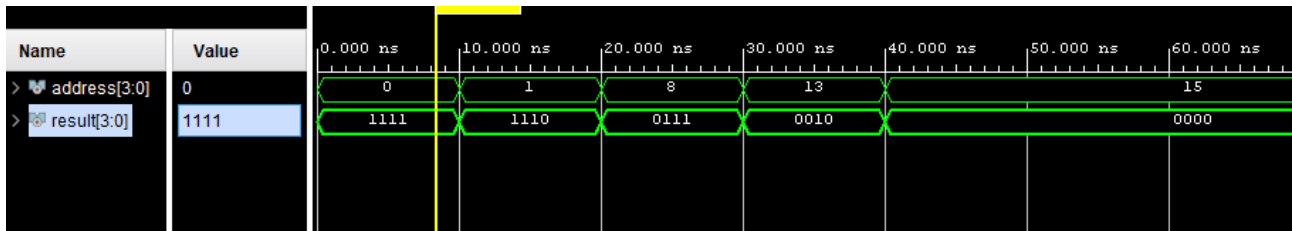
```

```

        assert result = "1111" report "Errore all'indirizzo 15" severity
error;

        wait;
    end process;
end architecture testbench;

```



### Sintesi su board di sviluppo

I primi 4 interruttori (dal pin **J15** al pin **R15**) della FPGA vengono usati per inserire l'indirizzo da cui leggere in memoria (**switch**), mentre i primi 4 led (da **H17** a **N14**) sono impiegati per visualizzare il risultato (**led**), ossia la parola letta trasformata da M.

```

##Switches
set_property -dict { PACKAGE_PIN J15    IOSTANDARD LVCMOS33 } [get_ports { switch[0]
}]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16    IOSTANDARD LVCMOS33 } [get_ports { switch[1]
}]; #IO_L3N_T0_QS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13    IOSTANDARD LVCMOS33 } [get_ports { switch[2]
}]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15    IOSTANDARD LVCMOS33 } [get_ports { switch[3]
}]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
## LEDs
set_property -dict { PACKAGE_PIN H17    IOSTANDARD LVCMOS33 } [get_ports { led[0]
}]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15    IOSTANDARD LVCMOS33 } [get_ports { led[1]
}]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13    IOSTANDARD LVCMOS33 } [get_ports { led[2]
}]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14    IOSTANDARD LVCMOS33 } [get_ports { led[3]
}]; #IO_L8P_T1_D11_14 Sch=led[3]

```

## Capitolo 2: Reti Sequenziali Elementari

### Esercizio 3: Riconoscitore di sequenze

#### Progetto e architettura

Un riconoscitore di sequenza ha come obiettivo quello di individuare determinati pattern di segnali di ingresso, ovvero specifiche sequenze di '0' e '1'. Una macchina di questo tipo può essere facilmente modellata come un automa a stati finiti di Mealy.

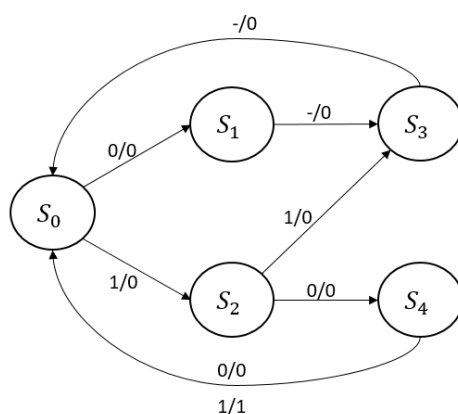
Il progetto richiede che sia riconosciuta la sequenza "101" e che la modalità sia determinata da un segnale (**M**). In particolare, per  $M=0$  la macchina osserva il segnale in input (**i**) alla ricerca di sequenze non sovrapposte e per  $M=1$  vengono invece ricercata sequenze parzialmente sovrapposte. Ad esempio, per la successione di ingressi "00010101000", nel primo caso si identifica una sola sequenza "101", nel secondo caso due.

Il modulo **riconoscitore** ha quattro segnali di ingresso: **i** (dato di ingresso), **RST** (segnale di reset), **CLK** (segnale di clock), e **M** (segnale di controllo). L'uscita del modulo è rappresentata dal segnale **Y**.

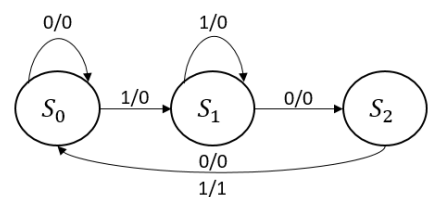
La architettura comportamentale del codice implementa la logica del riconoscitore attraverso due process: un processo (stato\_uscita) la cui sensitivity list include **i**, **M** e il segnale **stato\_corrente** che calcola le funzioni di transizione dello stato e dell'uscita, ovvero aggiorna opportunamente **stato\_prossimo** e **Y** in base agli attuali valori degli altri segnali; l'altro processo (mem) è sensibile al solo segnale di clock e se il reset è alto porta il sistema nello stato iniziale altrimenti se si è visto un fronte di salita di CLK si modifica stato\_corrente mettendolo uguale a stato\_prossimo. Si osservi che il reset è, in questo contesto, sincrono.

Sono stati identificati cinque stati potenziali, chiamati come **S0**, **S1**, **S2**, **S3** e **S4**; tuttavia, il significato attribuito a ciascuno di questi stati, così come il numero effettivamente utilizzato, è strettamente dipendente dalla modalità **M**. Successivamente, vengono presentati i diagrammi a stati del progetto per entrambe le possibili tecniche di riconoscimento.

Non sovrapposta ( $M=0$ )



Parzialmente sovrapposta ( $M=1$ )



#### Implementazione

**riconoscitore.vhd**

```
library IEEE;
```



```

use IEEE.STD_LOGIC_1164.ALL;

entity riconoscitore is
    port( i: in std_logic;
          RST, CLK: in std_logic;
          M : in std_logic;
          Y: out std_logic
        );
end riconoscitore;

architecture Behavioral of riconoscitore is

    type stato is (S0, S1, S2, S3, S4);

    signal stato_corrente : stato := S0;
    signal stato_prossimo : stato;

begin

    stato_uscita: process(stato_corrente, i, M)
    begin
        if(M'event) then
            stato_prossimo <= S0;
            Y <= '0';
        elsif(M='1') then
            case stato_corrente is
                when S0 =>
                    if( i = '0' ) then
                        stato_prossimo <= S0;
                        Y <= '0';
                    else
                        stato_prossimo <= S1;
                        Y <= '0';
                    end if;
                when S1 =>
                    if( i = '0' ) then
                        stato_prossimo <= S2;
                        Y <= '0';
                    else
                        stato_prossimo <= S1;
                        Y <= '0';
                    end if;
                when S2 =>
                    if( i = '0' ) then
                        stato_prossimo <= S0;
                        Y <= '0';
                    else
                        stato_prossimo <= S0;
                        Y <= '1';
                    end if;
            end case;
        end if;
    end process stato_uscita;
end

```

```

        when others =>
            stato_prossimo <= S0;
            Y <= '0';
    end case;
    elsif(M='0') then
    case stato_corrente is
        when S0 =>
            if( i = '0' ) then
                stato_prossimo <= S1;
                Y <= '0';
            else
                stato_prossimo <= S2;
                Y <= '0';
            end if;
        when S1 =>
            stato_prossimo <= S3;
            Y <= '0';
        when S3 =>
            stato_prossimo <= S0;
            Y <= '0';
        when S2 =>
            if( i = '0' ) then
                stato_prossimo <= S4;
                Y <= '0';
            else
                stato_prossimo <= S3;
                Y <= '0';
            end if;
        when S4 =>
            if( i = '0' ) then
                stato_prossimo <= S0;
                Y <= '0';
            else
                stato_prossimo <= S0;
                Y <= '1';
            end if;
        when others => stato_prossimo <= S0;
            Y <= '0';
    end case;
    end if;
end process;

mem: process (CLK)
begin
    if( CLK'event and CLK = '1' ) then
        if( RST = '1' ) then
            stato_corrente <= S0;
        else
            stato_corrente <= stato_prossimo;
        end if;
    end if;
end process;

```

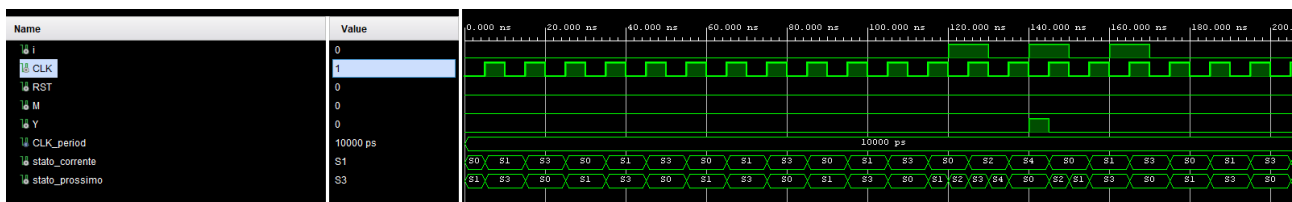
```
    end if;
end process;

end Behavioral;
```

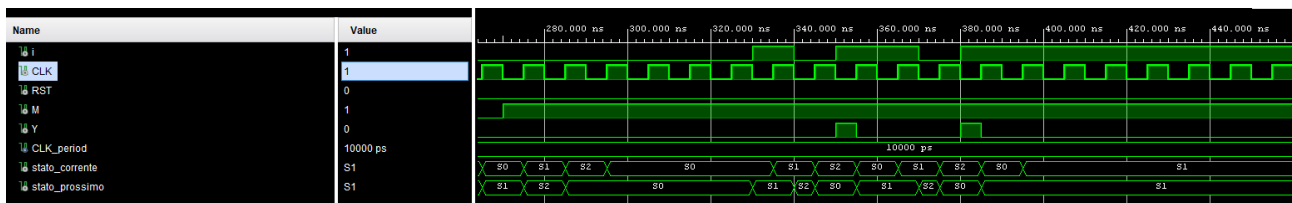
## Simulazione

Il testbench è stato progettato con l'obiettivo di simulare il comportamento del file "riconscitore.vhd"; esso include un process che genera il clock (**CLK\_process**) e uno che fornisce valori in ingresso, stimoli, a un'istanza del "design under test" (**stim\_proc**).

Durante la prima fase  $M$  è basso e viene fatto variare il valore del segnale  $i$  al fine di testare le transizioni di stato e osservare l'output del sistema.



Successivamente, il segnale di controllo M viene impostato su '1' e vengono forniti nuovi stimoli a i per studiare il circuito anche in questa modalità.



Come si può vedere, in entrambi i casi l'uscita si alza solo quando viene riconosciuta una sequenza "101" secondo l'approccio scelto e i passaggi di stato sono coerenti con la descrizione dell'automa precedentemente presentato.

**riconoscitore tb.vhd**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity riconoscitore_tb is
end riconoscitore_tb;

architecture Behavioral of riconoscitore_tb is

component riconoscitore
    port (
        i: in std_logic;
        RST, CLK: in std_logic;
        M : in std_logic;
        Y: out std_logic
    );
```

```

    end component;

    signal i : std_logic := '0';
    signal CLK : std_logic := '0';
    signal RST : std_logic := '0';
    signal M : std_logic := '0';

    signal Y : std_logic;

    constant CLK_period : time := 10 ns;

BEGIN
    dut: riconoscitore port map(
        i => i,
        CLK => CLK,
        RST => RST,
        M => M,
        Y => Y
    );

    CLK_process : process
    begin
        CLK <= '0';
        wait for CLK_period/2;
        CLK <= '1';
        wait for CLK_period/2;
    end process;

    stim_proc: process
    begin
        wait for 100 ns;
        M<='0';
        i<='0';
        wait for 10 ns;
        i<='0';
        wait for 10 ns;
        i<='1';
        wait for 10 ns;
        i<='0';
        wait for 10 ns;
        i<='1';
        wait for 10 ns;
        i<='0';
        wait for 10 ns;
        i<='1';
        wait for 10 ns;
        i<='0';
        wait for 100 ns;
        M<='1';
        i<='0';
    end process;

```

```

        wait for 50 ns;
        i<='0';
        wait for 10 ns;
        i<='1';
        wait for 10 ns;
        i<='0';
        wait for 10 ns;
        i<='1';
        wait for 10 ns;
        i<='1';
        wait for 10 ns;
        i<='0';
        wait for 10 ns;
        i<='1';
        wait;
    end process;
end;

```

---

#### *Sintesi su board di sviluppo*

Premendo il pulsante **B2** collegato al pin **P17** si seleziona la modalità M a partire dallo switch **S2** associato al pin **L16** e si mette a '0' l'ingresso; altrimenti se si preme il bottone **B1** del pin **N17** si campiona il valore dell'input dallo switch **J15**. Quando viene riconosciuto l'inserimento di una sequenza del tipo "101" si accende il led **H17**. Per resettare il sistema è sufficiente premere il pulsante al pin **M18**.

---

#### **riconoscitore\_board.vhd**

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity riconoscitore_board is
    Port (
        S1, S2, B1, B2 : in STD_LOGIC;
        RST, CLK : in STD_LOGIC;
        LED : out STD_LOGIC
    );
end riconoscitore_board;

architecture Behavioral of riconoscitore_board is
    signal i, M : STD_LOGIC := '0';
    signal Y : STD_LOGIC := '0';

    COMPONENT riconoscitore
        Port (
            i : in STD_LOGIC;
            RST, CLK : in STD_LOGIC;
            M : in STD_LOGIC;
            Y : out STD_LOGIC

```

```

    );
end COMPONENT;

begin
    riconoscitore_inst : riconoscitore
        port map (
            i=>i,
            RST=>RST,
            CLK=>CLK,
            M=>M,
            Y=>Y
        );

    process(CLK)
    begin
        if rising_edge(CLK) then
            if B2='1' then
                i<='0';
                M<=S2;
            elsif B1='1' then
                i<=S1;
            end if;
        end if;
    end process;

    LED <= Y;

end Behavioral;

```

---

## Clock signal

```
set_property -dict { PACKAGE_PIN E3  IOSTANDARD LVCMOS33 } [get_ports { CLK }];
```

```
#IO_L12P_T1_MRCC_35 Sch=clk100mhz
```

```
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK}];
```

##Switches

```
set_property -dict { PACKAGE_PIN J15  IOSTANDARD LVCMOS33 } [get_ports { S1 }];
```

```
#IO_L24N_T3_RS0_15 Sch=sw[0]
```

```
set_property -dict { PACKAGE_PIN L16  IOSTANDARD LVCMOS33 } [get_ports { S2 }];
```

```
#IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
```

## LEDs

```
set_property -dict { PACKAGE_PIN H17  IOSTANDARD LVCMOS33 } [get_ports { LED }];
```

```
#IO_L18P_T2_A24_15 Sch=led[0]
```

##Buttons

```
set_property -dict { PACKAGE_PIN N17  IOSTANDARD LVCMOS33 } [get_ports { B1 }];
```

```
#IO_L9P_T1_DQS_14 Sch=btnc
```

```
set_property -dict { PACKAGE_PIN M18  IOSTANDARD LVCMOS33 } [get_ports { RST }];  
#IO_L4N_T0_D05_14 Sch=btneu
```

```
set_property -dict { PACKAGE_PIN P17  IOSTANDARD LVCMOS33 } [get_ports { B2 }];  
#IO_L12P_T1_MRCC_14 Sch=btneu
```

## Esercizio 4: Shift Register

### Progetto e architettura

Il progetto prevede di implementare un registro a scorrimento di N bit in grado di shiftare il vettore a destra o sinistra di 1 o 2 posizioni. Si realizza il design sia da un livello di astrazione comportamentale che strutturale.

I parametri in ingresso al sistema includono, oltre al clock e al reset, il segnale **dir**, che indica se deve essere eseguito un left o un right shift, e il segnale **Y**, che specifica se il vettore in ingresso da N bit (**din**) deve essere caricato nello shift register in modalità parallela o se deve essere eseguito uno shift di 1 o 2 posizioni. Il generico **N** specifica la dimensione dello shift register; l'impiego di questo generico consente di parametrizzare il modulo in modo che esso possa essere riutilizzato per dati di diverse dimensioni. In ogni istante, i valori memorizzati nello shift register possono essere letti in parallelo tramite l'uscita **dout** da N bit.

Nell'architettura **Behavioral**, il comportamento del registro è descritto attraverso un processo sensibile al fronte di salita del segnale di clock (**rising\_edge(clk)**) e al segnale di reset (**rst**). Il reset è quindi asincrono.

L'operazione da eseguire è definita da una combinazione dei valori degli input Y e dir:

- Y = "00": caricamento parallelo
- Y = "01": shift di 1 posizione
  - dir = "0": shift a destra
  - dir = "1": shift a sinistra
- Y = "10" shift di 2 posizioni.
  - dir = "0": shift a destra
  - dir = "1": shift a sinistra

Per implementare ciò, vengono utilizzati sia il costrutto if...then...else che il costrutto case.

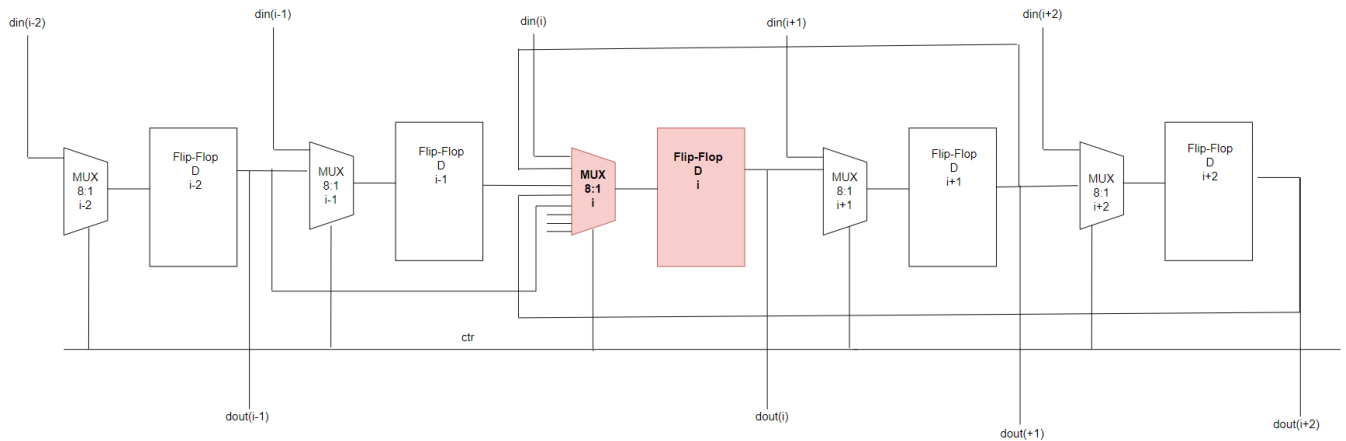
Nell'architettura **Structural**, il registro viene implementato per composizione di componenti di tipo **flip\_flop\_d** e **mux\_8\_1**. Il modulo **flip\_flop\_d** rappresenta un flip-flop D edge-triggered sul fronte di salita, mentre **mux\_8\_1** rappresenta un multiplexer 8 a 1. Il progetto prevede l'uso di N multiplexer e flip-flop opportunamente interconnessi tra loro: ogni multiplexer, in base a un segnale di selezione comune a tutti, imposta uno specifico bit in ingresso al flip-flop che lo segue.

Nella prossima immagine sono stati evidenziati soltanto i tipici collegamenti di un i-esimo multiplexer/flip-flop, con "i" compreso tra 2 e N-3. Grazie alla forte ripetizione, l'istanziamento di queste componenti nello shifter è svolta attraverso costrutti **for...generate**. Invece la realizzazione dei primi due stadi (i=0 e i=1) e degli ultimi due (i=N-2 e i=N-1) deve essere fatta a parte. Lo shift register così progettato richiede, quindi, che N sia come minimo 4.

A seconda del segnale di selezione condiviso **ctr**, il cui valore dipende da rst, Y e dir, l'ingresso del flip flop D i-esimo è uno dei seguenti:

- **din(i)** (nel caso del caricamento parallelo)
- **flip\_flop\_out(i+1)** (nel caso del Left Shift di 1 posizione)
- **flip\_flop\_out(i-1)** (nel caso del Right Shift di 1 posizione)
- **flip\_flop\_out(i+2)** (nel caso del Left Shift di 2 posizioni)
- **flip\_flop\_out(i-2)** (nel caso del Right Shift di 2 posizioni)





I porti dell'entity **mux\_8\_1** sono rappresentati da 8 bit in ingresso (da **input\_0** a **input\_7**), un segnale di controllo da 3 bit (**control**), sempre in input, e da un bit in uscita (**output**). L'architettura è comportamentale e include un processo sensibile ai cambiamenti degli input. Utilizzando una struttura case, il processo seleziona l'output in base al valore del segnale di controllo; ad esempio, se il segnale di controllo è "000", l'output sarà input\_0, e così via.

L'entity **flip\_flop\_d** presenta come ingressi i segnali **clk**, **reset** e **d** e come uscita il bit **q**. Nell'architecture "Behavioral", c'è un processo la cui sensitivity list è composta dal segnale di clock (clk) e da quello di reset (reset). Se il segnale di reset è '1', l'uscita viene impostata a '0'; in caso contrario, all'evento di fronte di salita (**rising\_edge**) di clk, l'uscita assume il valore dell'input d. Il flip-flop D edge-triggered rappresenta uno degli elementi di memoria (1 bit) più comunemente usati nei circuiti digitali.

### Implementazione

#### shift\_register.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity shift_register is
    generic (
        N : integer := 8
    );
    port (
        clk      : in  std_logic;
        rst      : in  std_logic;
        dir      : in  std_logic; -- 0: destra, 1: sinistra
        Y        : in  std_logic_vector(1 downto 0); -- 00: load, 01: shifta di 1
        din      : in  std_logic_vector(N-1 downto 0);
        dout     : out std_logic_vector(N-1 downto 0)
    );
end entity shift_register;
```

architecture Behavioral of shift\_register is

```
    signal reg : std_logic_vector(N-1 downto 0);
```

```
begin
```

```
    process(clk, rst)
```

```
    begin
```

```
        if rst = '1' then
```

```
            reg <= (others => '0');
```

```
        elsif rising_edge(clk) then
```

```
            case Y is
```

```
                when "00" => -- load
```

```
                    reg <= din;
```

```
                when "01" => -- una posizione
```

```
                    case dir is
```

```
                        when '0' => reg <= '0' & reg(N-1 downto 1); -- destra
```

```
                        when '1' => reg <= reg(N-2 downto 0) & '0'; -- sinistra
```

```
                        when others => reg <= (others=>'0');
```

```
                    end case;
```

```
                when "10" => -- due posizioni
```

```
                    case dir is
```

```
                        when '0' => reg <= "00" & reg(N-1 downto 2); -- destra
```

```
                        when '1' => reg <= reg(N-3 downto 0) & "00"; -- sinistra
```

```
                        when others => reg <= (others=>'0');
```

```
                    end case;
```

```
                when others =>
```

```
                    reg <= (others=>'0');
```

```
            end case;
```

```
        end if;
```

```
    end process;
```

```
    dout <= reg;
```

```
end architecture Behavioral;
```

architecture Structural of shift\_register is

```
    component flip_flop_d
```

```
    port ( clk      : in STD_LOGIC;
```

```
          reset    : in STD_LOGIC;
```

```
          d        : in STD_LOGIC;
```

```
          q        : out STD_LOGIC);
```

```
    end component;
```

```
    component mux_8_1
```

```
    port (
```

```
        input_0 : in STD_LOGIC;
```

```
        input_1 : in STD_LOGIC;
```

```
        input_2 : in STD_LOGIC;
```

```
        input_3 : in STD_LOGIC;
```

```

    input_4 : in STD_LOGIC;
    input_5 : in STD_LOGIC;
    input_6 : in STD_LOGIC;
    input_7 : in STD_LOGIC;
    control : in STD_LOGIC_VECTOR(2 downto 0);
    output : out STD_LOGIC
);
end component;

signal mux_out: std_logic_vector(N-1 downto 0) ;
signal flip_flop_out: std_logic_vector(N-1 downto 0);
signal ctr: std_logic_vector(2 downto 0);
signal dir_new: std_logic;

```

begin

```

    ctr <= "101" when rst='1' else
           "000" when Y="00" else
           "001" when dir='0' and Y="01" else
           "010" when dir='1' and Y="01" else
           "011" when dir='0' and Y="10" else
           "100" when dir='1' and Y="10";

```

```

    flip_flop_gen: for i in 0 to N-1 generate
        flip_flop: flip_flop_d port map(
            clk => clk,
            reset => rst,
            d => mux_out(i),
            q => flip_flop_out(i)
        );
    end generate flip_flop_gen;

```

```

    mux_0: mux_8_1 port map(
        input_0 => din(0), -- caricamento
        input_1 => flip_flop_out(1), -- shift di una posizione a sinistra
        input_2 => '0', -- shift di una posizione a destra
        input_3 => flip_flop_out(2), -- shift di due posizioni a sinistra
        input_4 => '0', -- shift di due posizioni a destra
        input_5 => '0', -- reset
        input_6 => '0', -- non usato
        input_7 => '0', -- non usato
        control => ctr,
        output => mux_out(0)
    );

```

```

    mux_1: mux_8_1 port map(
        input_0 => din(1),
        input_1 => flip_flop_out(2),
        input_2 => flip_flop_out(0),
        input_3 => flip_flop_out(3),
        input_4 => '0',

```

```

    input_5 => '0',
    input_6 => '0',
    input_7 => '0',
    control => ctr,
    output => mux_out(1)
);

mux_N_1: mux_8_1 port map(
    input_0 => din(N-1),
    input_1 => '0',
    input_2 => flip_flop_out(N-2),
    input_3 => '0',
    input_4 => flip_flop_out(N-3),
    input_5 => '0',
    input_6 => '0',
    input_7 => '0',
    control => ctr,
    output => mux_out(N-1)
);

mux_N_2: mux_8_1 port map(
    input_0 => din(N-2),
    input_1 => flip_flop_out(N-1),
    input_2 => flip_flop_out(N-3),
    input_3 => '0',
    input_4 => flip_flop_out(N-4),
    input_5 => '0',
    input_6 => '0',
    input_7 => '0',
    control => ctr,
    output => mux_out(N-2)
);

mux_gen: for i in 2 to N-3 generate
    mux_interni: mux_8_1 port map(
        input_0 => din(i),
        input_1 => flip_flop_out(i+1),
        input_2 => flip_flop_out(i-1),
        input_3 => flip_flop_out(i+2),
        input_4 => flip_flop_out(i-2),
        input_5 => '0',
        input_6 => '0',
        input_7 => '0',
        control => ctr,
        output => mux_out(i)
    );
end generate mux_gen;

dout <= flip_flop_out;

```

```
end Structural;
```

---

### **mux\_8\_1.vhd**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mux_8_1 is
    Port (
        input_0 : in STD_LOGIC;
        input_1 : in STD_LOGIC;
        input_2 : in STD_LOGIC;
        input_3 : in STD_LOGIC;
        input_4 : in STD_LOGIC;
        input_5 : in STD_LOGIC;
        input_6 : in STD_LOGIC;
        input_7 : in STD_LOGIC;
        control : in STD_LOGIC_VECTOR(2 downto 0);
        output : out STD_LOGIC
    );
end mux_8_1;

architecture Behavioral of mux_8_1 is

begin

    process (input_0, input_1, input_2, input_3, input_4, input_5, input_6,
input_7, control)
    begin
        case control is
            when "000" =>
                output <= input_0;
            when "001" =>
                output <= input_1;
            when "010" =>
                output <= input_2;
            when "011" =>
                output <= input_3;
            when "100" =>
                output <= input_4;
            when "101" =>
                output <= input_5;
            when "110" =>
                output <= input_6;
            when "111" =>
                output <= input_7;
            when others =>
```

```

        output <= '0';
    end case;
end process;

end Behavioral;

```

---

#### flip\_flop\_d.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity flip_flop_d is Port (
    clk    : in STD_LOGIC;
    reset   : in STD_LOGIC;
    d       : in STD_LOGIC;
    q       : out STD_LOGIC);
end flip_flop_d;

architecture Behavioral of flip_flop_d is

    signal q_out : STD_LOGIC;

begin
    process(clk, reset)
    begin
        if reset = '1' then
            q_out <= '0';
        elsif rising_edge(clk) then
            q_out <= d;
        end if;
    end process;

    q <= q_out;

end Behavioral;

```

---

#### Simulazione

Si sono creati due testbench, uno dedicato alla simulazione dello shift register realizzato a partire dall'architecture comportamentale (**shift\_register\_behavioral\_tb**) e uno per testare l'architettura strutturale (**shift\_register\_structural\_tb**). Sono stati scelti due diversi valori per il parametro N (8 nel primo caso, 12 nel secondo).

---

#### shift\_register\_structural\_tb.vhd

```

library IEEE;

```

```

use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity shift_register_structural_tb is
end entity shift_register_structural_tb;

architecture testbench of shift_register_structural_tb is
    constant N : integer := 12;

    signal clk      : std_logic := '0';
    signal rst      : std_logic := '0';
    signal shift_dir: std_logic := '0';
    signal shift_amp: std_logic_vector(1 downto 0) := "00";
    signal din      : std_logic_vector(N-1 downto 0) := (others => '0');
    signal dout     : std_logic_vector(N-1 downto 0) := (others => '0');
    constant clk_period : time := 10 ns;

    component shift_register
        generic (
            N : integer := N
        );
        port (
            clk      : in  std_logic;
            rst      : in  std_logic;
            shift_dir: in  std_logic;
            shift_amp: in  std_logic_vector(1 downto 0);
            din      : in  std_logic_vector(N-1 downto 0);
            dout     : out std_logic_vector(N-1 downto 0)
        );
    end component;

begin
    dut: entity work.shift_register(Structural)
        generic map (N=>N)
        port map (
            clk      => clk,
            rst      => rst,
            dir      => shift_dir,
            Y        => shift_amp,
            din      => din,
            dout     => dout
        );

    CLK_process :process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

```

```
end process;
```

```
process
```

```
begin
```

```
    din <= "110101100111";
```

```
    shift_amp <= "00"; -- load
```

```
    wait for 10 ns;
```

```
    shift_dir <= '0'; -- shift a destra di...
```

```
    shift_amp <= "01"; -- ...1 posizione
```

```
    wait for 30 ns;
```

```
    shift_amp <= "10"; -- ...2 posizioni
```

```
    wait for 30 ns;
```

```
    din <= "010101110010";
```

```
    shift_amp <= "00"; -- load
```

```
    wait for 10 ns;
```

```
    shift_dir <= '1'; -- shift a sinistra di...
```

```
    shift_amp <= "01"; -- ...1 posizione
```

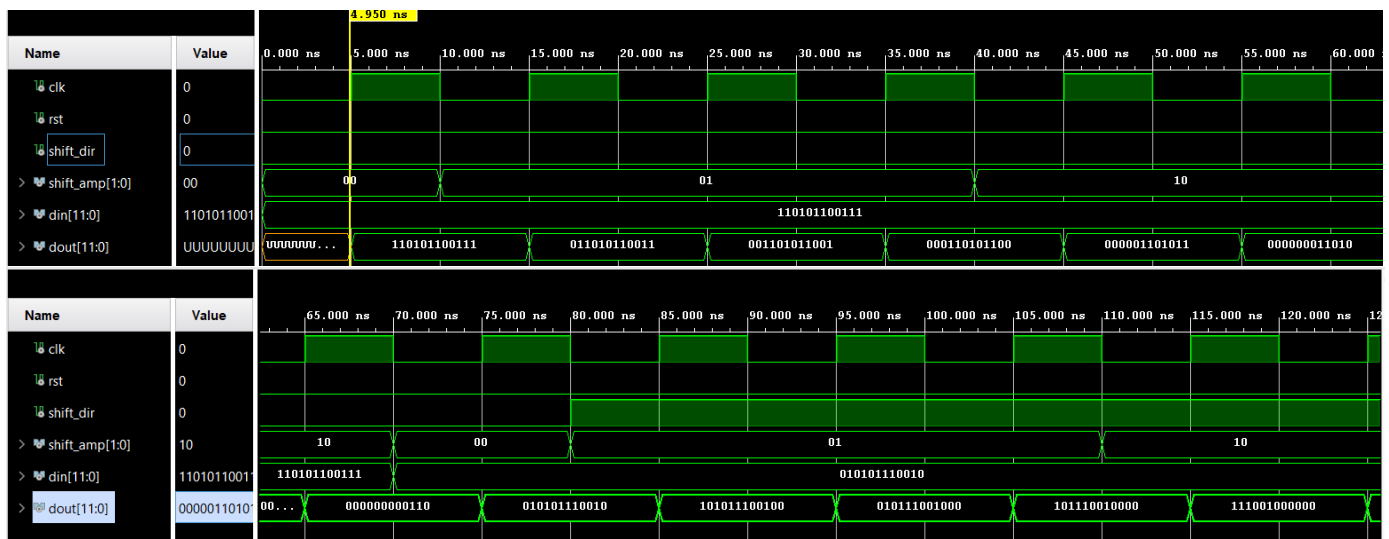
```
    wait for 30 ns;
```

```
    shift_amp <= "10"; -- ...2 posizioni
```

```
    wait;
```

```
end process;
```

```
end architecture testbench;
```



## shift\_register\_behavioral\_tb.vhd

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
```

```
use IEEE.STD_LOGIC_ARITH.ALL;
```

```
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity shift_register_behavioral_tb is
```



```

end entity shift_register_behavioral_tb;

architecture testbench of shift_register_behavioral_tb is
    constant N : integer := 8;

    signal clk      : std_logic := '0';
    signal rst      : std_logic := '0';
    signal shift_dir: std_logic := '0';
    signal shift_amp: std_logic_vector(1 downto 0) := "00";
    signal din      : std_logic_vector(N-1 downto 0) := (others => '0');
    signal dout     : std_logic_vector(N-1 downto 0) := (others => '0');
    constant clk_period : time := 10 ns;

    component shift_register
        generic (
            N : integer := N
        );
        port (
            clk      : in  std_logic;
            rst      : in  std_logic;
            shift_dir: in  std_logic;
            shift_amp: in  std_logic_vector(1 downto 0);
            din      : in  std_logic_vector(N-1 downto 0);
            dout     : out std_logic_vector(N-1 downto 0)
        );
    end component;

begin
    dut: entity work.shift_register(Behavioral)
        generic map (N=>N)
        port map (
            clk      => clk,
            rst      => rst,
            dir      => shift_dir,
            Y        => shift_amp,
            din      => din,
            dout     => dout
        );

    CLK_process :process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

    process
    begin

```

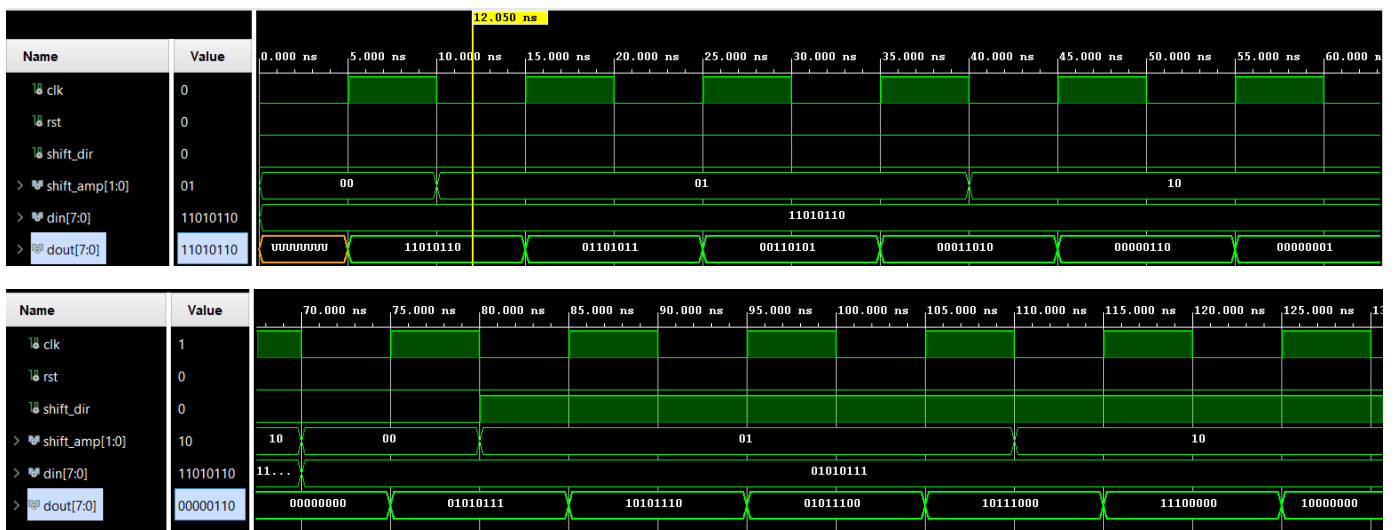
```

din <= "11010110";
shift_amp <= "00"; -- load
wait for 10 ns;
shift_dir <= '0'; -- shift a destra di...
shift_amp <= "01"; -- ...1 posizione
wait for 30 ns;
shift_amp <= "10"; -- ...2 posizioni
wait for 30 ns;

din <= "01010111";
shift_amp <= "00"; -- load
wait for 10 ns;
shift_dir <= '1'; -- shift a sinistra di...
shift_amp <= "01"; -- ...1 posizione
wait for 30 ns;
shift_amp <= "10"; -- ...2 posizioni
wait;
end process;

end architecture testbench;

```



## Esercizio 5: Cronometro

### Progetto e architettura

Per la realizzazione del cronometro, si è adottato un approccio strutturale nel quale tre contatori distinti, uno dedicato a monitorare i secondi, uno i minuti e un altro le ore, sono opportunamente connessi in uno schema seriale. Ogni contatore è realizzato come una specifica istanza di un'entità contatore il cui modulo è parametrizzato, ovvero si definisce il generico **N**. Per i contatori **sec\_counter** e **min\_counter**, **N** è stato impostato a 59 dato che esso rappresenta il massimo valore consentito per i secondi e i minuti, mentre per **ore\_counter**, **N** è 23 poichè questa è l'ora massima consentita nel sistema orario a 24 ore. Nel contatore (**contatore**) all'impulso di conteggio successivo al raggiungimento del picco il valore (**count**) si azzerava.

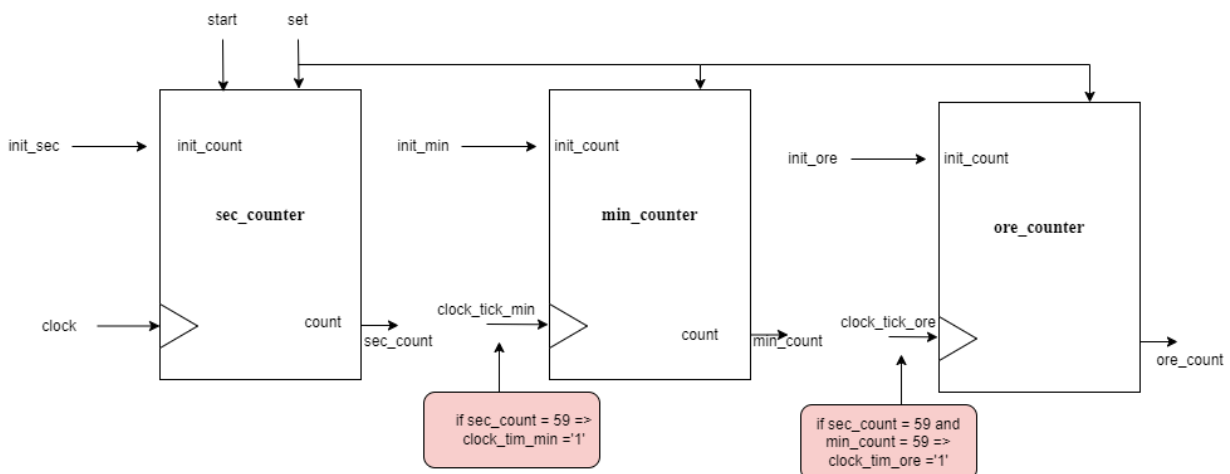
Per poter precaricare un valore di conteggio nei contatori si sfrutta un segnale di abilitazione **set**. Quando **set** è attivo, ogni contatore imposta il valore intero di conteggio a uno fornito dall'esterno (**init\_sec** per **sec\_counter**, **init\_min** per **min\_counter**, e **init\_ore** per **ore\_counter**).

Solo il primo contatore, ovvero quello dei secondi, prende in ingresso il segnale di sincronismo **clock**, tale che ad ogni colpo di clock **sec\_count** incrementi il suo valore.

**min\_counter**, invece, prende in ingresso come segnale di tempificazione il segnale **clock\_tick\_min** il quale, in modo sincrono a clock, si alza sono nel momento in cui il valore in uscita a **sec\_counter** è il massimo, ovvero 59, altrimenti è basso. In seguito a transizioni di **clock\_tick\_min** del tipo **0→1→0**, **min\_count** si incrementa di 1.

Analogamente **ore\_counter** ha come clock il segnale **clock\_tick\_ore**, che viene abilitato sono quando **sec\_count** e **min\_count** raggiunto entrambi il massimo valore di 59, altrimenti è '0'. Un impulso di questo tipo genera un incremento di **ore\_count**.

Di seguito è possibile visualizzare l'architettura del cronometro.



### Implementazione

**cronometro.vhd**

`library IEEE;`

```
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity cronometro is
```

```
    port (  
        clk : in std_logic;  
        reset : in std_logic;  
        set : in std_logic;  
        init_sec : in integer range 0 to 59;  
        init_min : in integer range 0 to 59;  
        init_ore : in integer range 0 to 23;  
        sec : out integer range 0 to 59;  
        min : out integer range 0 to 59;  
        ore : out integer range 0 to 23  
    );
```

```
end entity;
```

```
architecture Structural of cronometro is
```

```
    component contatore
```

```
        generic (  
            MAX_VALUE : integer := 59  
        );  
        port (  
            clk : in std_logic;  
            reset : in std_logic;  
            set : in std_logic;  
            init_count : in integer range 0 to MAX_VALUE;  
            count : out integer range 0 to MAX_VALUE  
        );
```

```
    end component;
```

```
    signal sec_count : integer range 0 to 59;  
    signal min_count : integer range 0 to 59;  
    signal ore_count : integer range 0 to 23;  
    signal clock_tick_min : std_logic := '0';  
    signal clock_tick_ore : std_logic := '0';
```

```
begin
```

```
    sec_counter : contatore  
        generic map (MAX_VALUE => 59)  
        port map (  
            clk => clk,  
            reset => reset,  
            set => set,  
            init_count => init_sec,  
            count => sec_count  
        );
```

```
    min_counter : contatore  
        generic map (MAX_VALUE => 59)  
        port map (  
            clk => clock_tick_min,
```

```

        reset => reset,
        set => set,
        init_count => init_min,
        count => min_count
    );

    ore_counter : contatore
        generic map (MAX_VALUE => 23)
        port map (
            clk => clock_tick_ore,
            reset => reset,
            set => set,
            init_count => init_ore,
            count => ore_count
        );

    process(clk, reset)
    begin
        if rising_edge(clk) and reset='0' then
            if sec_count = 59 then
                clock_tick_min <= '1';
            else
                clock_tick_min <= '0';
                clock_tick_ore <= '0';
            end if;

            if min_count = 59 and sec_count = 59 then
                clock_tick_ore <= '1';
            end if;

        end if;
    end process;

    sec <= sec_count;
    min <= min_count;
    ore <= ore_count;
end architecture;

```

---

## contatore.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity contatore is
    generic (
        MAX_VALUE : integer := 59
    );
    port (
        clk : in std_logic;
        reset : in std_logic;

```

```

        set : in std_logic;
        init_count : in integer range 0 to MAX_VALUE;
        count : out integer range 0 to MAX_VALUE
    );
end entity;

architecture Behavioral of contatore is

    signal counter : integer range 0 to MAX_VALUE :=0;

begin

    process(clk, reset, set)
    begin
        if reset = '1' then
            counter <= 0;
        elsif set='1' then
            counter <= init_count;
        elsif rising_edge(clk) then
            if counter < MAX_VALUE then
                counter <= counter + 1;
            else
                counter <= 0;
            end if;
        end if;
    end process;

    count <= counter;

end architecture;

```

---

### *Simulazione*

---

#### **cronometro\_tb.vhd**

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity cronometro_tb is
end entity;

architecture tb of cronometro_tb is
    signal clk : std_logic := '0';
    signal reset : std_logic := '0';
    signal set : std_logic := '0';
    signal init_sec : integer := 0;

```

```

signal init_min : integer := 0;
signal init_ore : integer := 0;
signal sec : integer;
signal min : integer;
signal ore : integer;
begin
    crono_metro : entity work.cronometro
        port map (
            clk => clk,
            reset => reset,
            set => set,
            init_sec => init_sec,
            init_min => init_min,
            init_ore => init_ore,
            sec => sec,
            min => min,
            ore => ore
        );

    process
    begin
        wait for 5 ns;
        clk <= not clk;
    end process;

    process
    begin
        init_sec <= 50;
        init_min <= 59;
        init_ore <= 4;
        set <= '1';
        wait for 10 ns;
        set <= '0';

        wait for 200 ns;

        reset <= '1';
        wait for 10 ns;
        reset <= '0';

        wait for 650 ns;

        init_sec <= 58;
        init_min <= 59;
        init_ore <= 23;
        set <= '1';
        wait for 10 ns;
        set <= '0';

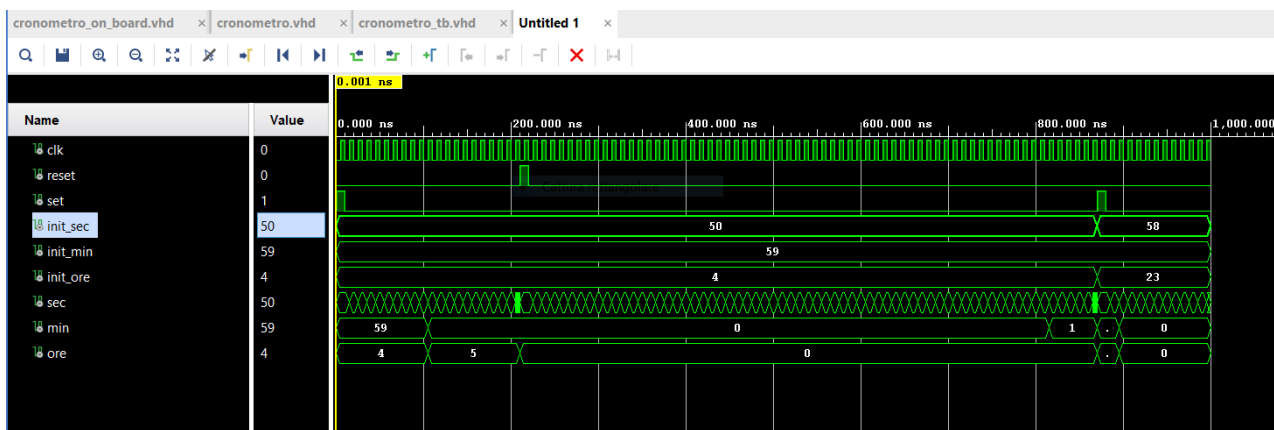
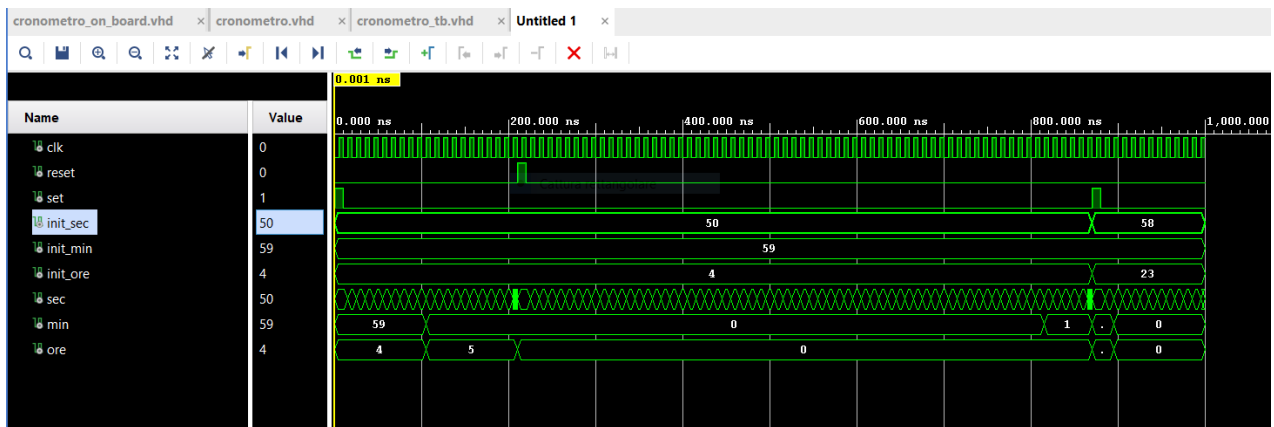
        wait for 100 ns;
    end process;
end;

```

```

wait;
end process;
end tb;

```



### Sintesi su board di sviluppo

La sintesi sul board Nexys A7-100T richiede la scrittura di un modulo dedicato. L'entity **cronometro\_on\_board** presenta diverse porte di input e output necessarie per interfacciarsi con il board di sviluppo:

- **reset** è collegato al bottone di pin P17.
- **input** è un vettore di 6 bit collegato ai primi 6 switch da **J15** a **T18**, i quali vengono utilizzati per inserire i dati in ingresso sec\_init, min\_init e ore\_init. In realtà, per quanto riguarda le ore, sono sufficienti i primi 5 switch.
- **input\_ore (U12)**, **input\_min (U11)**, **input\_sec (V10)** sono associati a switch; quando uno solo di essi è alto, si fa l'acquisizione da "input" del valore iniziale associato.
- **input\_count (bottone M18)** è il segnale di set per il caricamento dei valori iniziali nel cronometro.
- **stop** è associato al bottone P18; alla sua pressione si esegue l'acquisizione di un intertempo.
- Le prime sei cifre del display a 7 segmenti sono utilizzate per visualizzare il tempo corrente nel formato del tipo "23.59.59", compreso di dots.
- **view** è connesso al bottone N17 il quale, quando è premuto, causa la visualizzazione sul display di un intertempo precedentemente catturato invece che delle uscite del cronometro.

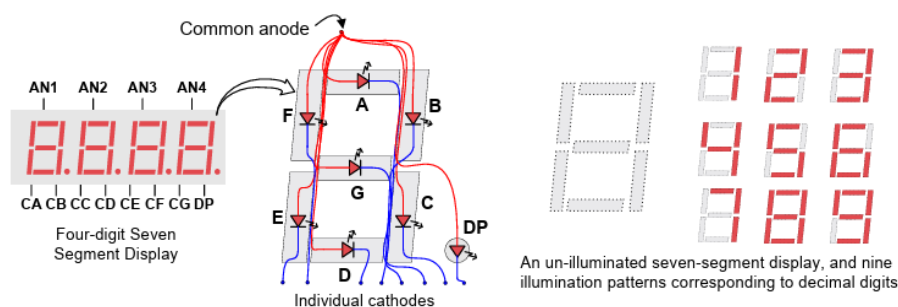


Per poter eseguire l'acquisizione e la visualizzazione degli intertempi è stato necessario progettare un nuovo modulo ossia una memoria MEM che potesse contenere 8 word da 17 bit (5 per l'ora, 6 per i minuti, 6 per i secondi).

MEM prende in ingresso come clock la **OR** tra il segnale stop e view così che essa si attivi sia per le operazioni di lettura che scrittura. Per scandire gli indirizzi a cui accedere si sono istanziati due contatori di tipo "**counter\_mod8.vhd**": **counter\_addr\_0** che fornisce l'indirizzo in cui scrivere il nuovo intertempo catturato e **counter\_addr\_1** per fornisce l'indirizzo da cui leggere il prossimo intertempo.

Quindi, se si preme stop viene scritta nella cella di memoria all'indirizzo **address\_0** il valore corrente del cronometro dato dal segnale **tempo** da 17 bit (concatenazione vettori **ore**, **min** e **sec**); invece se viene pressato view si visualizza sul display il valore dell'intertempo all'indirizzo **address\_1**.

Il display a 7 segmenti è chiamato così perché ogni cifra del display è composta da 7 led che possono essere individualmente accesi per un totale di 128 possibili combinazioni. Gli anodi dei 7 segmenti sono tutti collegati tra loro mentre i catodi sono separati. I catodi di segmenti "simili" su tutte le cifre del display sono collegati a sette nodi di circuito denominati da CA a CG. Il circuito di controllo del display (**display\_seven\_segments**) può essere utilizzato per visualizzare un numero a 8 cifre su questo display. Questo circuito pilota i segnali di anodo e i modelli di catodo corrispondenti di ogni cifra in una successione continua e ripetitiva, a una frequenza di aggiornamento più rapida della risposta dell'occhio umano.



Il **display\_seven\_segments** è composto da un **counter\_mod8**, **cathodes\_manager**, **andoes\_manager** e un **clock\_filter**. L'uscita del **clock\_filter**, ovvero **clock\_filter\_out** va in ingresso come segnale di abilitazione al contatore\_mod8. Il valore in uscita dal contatore a sua volta va in ingresso al gestore dei catodi e anodi per selezionare il segmento da accendere. Sarà **cathodes\_manager** a prendere in ingresso il valore temporale (**value**) da mostrare. **value** può essere l'uscita del cronometro oppure un intertempo a seconda del caso.

Si sono introdotti degli alias per suddividere il valore in ingresso al display: **sec** rappresenta i primi 6 bit, **min** comprende i bit dal sesto all'undicesimo, e **ore** spazia dai bit 12 ai 16. Sono, inoltre, stati definiti 6 segnali che rappresentano le cifre da visualizzare: **cifra\_0** e **cifra\_1** sono quelle dei secondi, **cifra\_2** e **cifra\_3** sono quelle dei minuti, **cifra\_4** e **cifra\_5** sono quelle delle ore.

Per ottenere le due cifre dei secondi, si converte il vettore di bit **sec**, di tipo **std\_logic\_vector**, in un numero intero senza segno. Successivamente, per ricavare la cifra meno significativa è sufficiente calcolare il resto della divisione di questo numero per 10; invece il risultato della divisione dell'intero sempre per 10 rappresenta la cifra più significativa. Il processo verrà replicato per le cifre dei minuti e delle ore.

---

## MEM.vhd

```
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.numeric_std.all;
```

```

entity MEM is
    port (
        clk      : in  std_logic;
        address   : in  std_logic_vector(2 downto 0);
        read_write : in  std_logic;
        d_in      : in  std_logic_vector(16 downto 0);
        d_out     : out std_logic_vector(16 downto 0)
    );
end entity MEM;

architecture RTL of MEM is
    type MEMORY is array (0 to 7) of std_logic_vector(16 downto 0);
    signal mem_data : MEMORY := (others => (others => '0'));

begin
    process (clk)
    begin
        if rising_edge(clk) then
            if(read_write='1') then --scrivi
                mem_data(to_integer(unsigned(address))) <= d_in;
            else d_out<=mem_data(to_integer(unsigned(address)));
            end if;
        end if;
    end process;

end architecture RTL;

```

---

#### cronometro\_on\_board.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity cronometro_on_board is
    port (
        CLK : in std_logic;
        stop : in std_logic; -- pulsante per catturare gli intertempi
        view : in std_logic; -- pulsante per visualizzare gli intertempi
        input : in std_logic_vector(5 downto 0);
        input_sec : in std_logic;
        input_min : in std_logic;
        input_ore : in std_logic;
        anodes_out : out  STD_LOGIC_VECTOR (7 downto 0);
        cathodes_out : out STD_LOGIC_VECTOR (7 downto 0);
        input_count : in std_logic;
        reset : in std_logic
    );
end cronometro_on_board;

architecture Behavioral of cronometro_on_board is

```

```

component display_seven_segments
generic(
    CLKIN_freq : integer := 100000000;
    CLKOUT_freq : integer := 500
);
port(
    CLK : IN std_logic;
    RST : IN std_logic;
    VALUE : IN std_logic_vector(16 downto 0);
    ENABLE : IN std_logic_vector(7 downto 0);
    DOTS : IN std_logic_vector(7 downto 0);
    ANODES : OUT std_logic_vector(7 downto 0);
    CATHODES : OUT std_logic_vector(7 downto 0)
);
end component;

component cronometro is
port (
    clk : in std_logic;
    reset : in std_logic;
    set : in std_logic;
    init_sec : in integer range 0 to 59;
    init_min : in integer range 0 to 59;
    init_ore : in integer range 0 to 23;
    sec : out integer range 0 to 59;
    min : out integer range 0 to 59;
    ore : out integer range 0 to 23
);
end component;

component counter_mod8 is
port (
    clock : in STD_LOGIC;
    reset : in STD_LOGIC;
    enable : in STD_LOGIC;
    counter : out STD_LOGIC_VECTOR (2 downto 0)
);
end component;

component MEM is
port (
    clk : in std_logic;
    address : in std_logic_vector(2 downto 0);
    d_in : in std_logic_vector(16 downto 0);
    read_write : in std_logic; -- se 0 basso di legge, se 1 alto si
    d_out : out std_logic_vector(16 downto 0)
);
end component;

```

scrive

```

    signal value : std_logic_vector(16 downto 0); -- ingresso a
display_seven_segments
    signal tempo : std_logic_vector(16 downto 0); -- uscita di cronometro
    signal sec, min, ore : integer := 0;
    signal sec_init, min_init, ore_init: integer := 0;
    signal counter : integer := 0;
    signal clock : std_logic := '0';
    signal address_0, address_1, address : std_logic_vector(2 downto 0);
    signal intertempo : std_logic_vector(16 downto 0);
    signal stop_view : std_logic;
    signal r_w : std_logic; -- read o write?

begin

    new_clk : process(CLK) -- realizza un clock con periodo di 1 sec a
partire da CLK
    begin
        if rising_edge(CLK) then
            counter <= counter + 1;
            if counter = (100_000_000)/2 then
                counter <= 0;
                clock <= not clock;
            end if;
        end if;
    end process;

    tempo <= std_logic_vector(to_unsigned(ore,5)) &
std_logic_vector(to_unsigned(min,6)) & std_logic_vector(to_unsigned(sec,6));

    cronom : cronometro port map (
        clk => clock,
        reset => reset,
        set => input_count,
        init_sec => sec_init,
        init_min => min_init,
        init_ore => ore_init,
        sec => sec,
        min => min,
        ore => ore );

    init_counter : process(CLK) -- per settare il conteggio iniziale
begin
    if(input_sec='1' and input_min='0' and input_ore='0') then
        sec_init <= to_integer(unsigned(input));
    elsif(input_sec='0' and input_min='1' and input_ore='0') then
        min_init <= to_integer(unsigned(input));
    elsif(input_sec='0' and input_min='0' and input_ore='1') then
        ore_init <= to_integer(unsigned(input(4 downto 0)));
    end if;
end process;

```

```

end process;

counter_addr_0 : counter_mod8 port map ( -- per ricavare l'indirizzo nel
quale scrivere il prossimo intertempo nella memoria
    clock => stop,
    reset => reset,
    enable => '1',
    counter => address_0
);

counter_addr_1 : counter_mod8 port map ( -- per ricavare l'indirizzo da
cui leggere un successivo intertempo, in precedenza scritto nella memoria
    clock => view,
    reset => reset,
    enable => '1',
    counter => address_1
);

stop_view <= stop or view;

address <= address_1 when view='1' else address_0;

gestione_intertempi : process(address)
begin
    if view='1' then
        r_w <= '0';
        value <= intertempo;
    else
        r_w <= '1';
        value <= tempo;
    end if;
end process;

intertempi_mem : MEM port map ( -- memoria che contiene gli intertempi
    clk => stop_view,
    address => address,
    read_write => r_w,
    d_in => tempo,
    d_out => intertempo
);

seven_segment_array: display_seven_segments
generic map(
    CLKIN_freq => 100000000,
    CLKOUT_freq => 500 )
port map(
    CLK => CLK,
    RST => reset,
    value => value,
    enable => "00111111",

```

```

        dots => "00010100",
        anodes => anodes_out,
        cathodes => cathodes_out
    );

end Behavioral;

```

---

## cathodes\_manager.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity cathodes_manager is
    Port ( counter : in  STD_LOGIC_VECTOR (2 downto 0);
          value : in  std_logic_vector(16 downto 0); --dato di mostrare
          sugli 8 display
          dots : in  STD_LOGIC_VECTOR (7 downto 0); --configurazione punti
          da accendere
          cathodes : out  STD_LOGIC_VECTOR (7 downto 0)); --sono i 7 catodi
più il punto
end cathodes_manager;

architecture Behavioral of cathodes_manager is

    constant zero    : std_logic_vector(6 downto 0) := "1000000";
    constant one     : std_logic_vector(6 downto 0) := "1111001";
    constant two     : std_logic_vector(6 downto 0) := "0100100";
    constant three   : std_logic_vector(6 downto 0) := "0110000";
    constant four    : std_logic_vector(6 downto 0) := "0011001";
    constant five    : std_logic_vector(6 downto 0) := "0010010";
    constant six     : std_logic_vector(6 downto 0) := "0000010";
    constant seven   : std_logic_vector(6 downto 0) := "1111000";
    constant eight   : std_logic_vector(6 downto 0) := "0000000";
    constant nine    : std_logic_vector(6 downto 0) := "0010000";

    alias sec is value (5 downto 0);
    alias min is value (11 downto 6);
    alias ore is value (16 downto 12);

    signal cifra_0 : std_logic_vector(3 downto 0);

```

```

signal cifra_1 : std_logic_vector(3 downto 0);
signal cifra_2 : std_logic_vector(3 downto 0);
signal cifra_3 : std_logic_vector(3 downto 0);
signal cifra_4 : std_logic_vector(3 downto 0);
signal cifra_5 : std_logic_vector(3 downto 0);

signal cathodes_for_digit : std_logic_vector(6 downto 0) := (others => '0');
signal nibble :std_logic_vector(3 downto 0) := (others => '0');
signal dot :std_logic := '0'; --stabilisce se il punto relativo alla cifra
visualizzata deve essere acceso o spento
                                --nota: dot=1 significa che deve essere acceso,
ma il segnale deve essere negato per andare sui catodi

begin

    cifra_0 <= std_logic_vector(to_unsigned((to_integer(unsigned(sec)) rem
10),4));
    cifra_1 <= std_logic_vector(to_unsigned((to_integer(unsigned(sec)) /
10),4));
    cifra_2 <= std_logic_vector(to_unsigned((to_integer(unsigned(min)) rem
10),4));
    cifra_3 <= std_logic_vector(to_unsigned((to_integer(unsigned(min)) /
10),4));
    cifra_4 <= std_logic_vector(to_unsigned((to_integer(unsigned(ore)) rem
10),4));
    cifra_5 <= std_logic_vector(to_unsigned((to_integer(unsigned(ore)) /
10),4));

-- questo processo multiplexa le cifre da mostrare
digit_switching: process(counter)

begin
    case counter is
        when "000" =>
            nibble <= cifra_0;
            dot <= dots(0);
        when "001" =>
            nibble <= cifra_1;
            dot <= dots(1);
        when "010" =>
            nibble <= cifra_2;
            dot <= dots(2);
        when "011" =>
            nibble <= cifra_3;
            dot <= dots(3);
        when "100" =>
            nibble <= cifra_4;
            dot <= dots(4);
        when "101" =>

```

```

        nibble <= cifra_5;
        dot <= dots(5);
    when "110" =>
        nibble <= "0000";
        dot <= dots(6);
    when "111" =>
        nibble <= "0000";
        dot <= dots(7);
    when others =>
        nibble <= (others => '0');
        dot <= '0';
    end case;
end process;

seven_segment_decoder_process: process(nibble)
begin
    case nibble is
        when "0000" => cathodes_for_digit <= zero;
        when "0001" => cathodes_for_digit <= one;
        when "0010" => cathodes_for_digit <= two;
        when "0011" => cathodes_for_digit <= three;
        when "0100" => cathodes_for_digit <= four;
        when "0101" => cathodes_for_digit <= five;
        when "0110" => cathodes_for_digit <= six;
        when "0111" => cathodes_for_digit <= seven;
        when "1000" => cathodes_for_digit <= eight;
        when "1001" => cathodes_for_digit <= nine;
        when others => cathodes_for_digit <= (others => '0');
    end case;
end process seven_segment_decoder_process;

cathodes <= (not dot)&cathodes_for_digit;

end Behavioral;

```

---

## Clock signal

```

set_property -dict { PACKAGE_PIN E3  IOSTANDARD LVCMOS33 } [get_ports { CLK }];
#IO_L12P_T1_MRCC_35 Sch=clk100mhz

```

```

create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK}];

```

##Switches

```

set_property -dict { PACKAGE_PIN J15  IOSTANDARD LVCMOS33 } [get_ports { input[0] }];
#IO_L24N_T3_RS0_15 Sch=sw[0]

```

```

set_property -dict { PACKAGE_PIN L16  IOSTANDARD LVCMOS33 } [get_ports { input[1] }];
#IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]

```

```

set_property -dict { PACKAGE_PIN M13  IOSTANDARD LVCMOS33 } [get_ports { input[2] }];
#IO_L6N_T0_D08_VREF_14 Sch=sw[2]

```



```
set_property -dict { PACKAGE_PIN R15  IOSTANDARD LVCMOS33 } [get_ports { input[3] }];  
#IO_L13N_T2_MRCC_14 Sch=sw[3]
```

```
set_property -dict { PACKAGE_PIN R17  IOSTANDARD LVCMOS33 } [get_ports { input[4] }];  
#IO_L12N_T1_MRCC_14 Sch=sw[4]
```

```
set_property -dict { PACKAGE_PIN T18  IOSTANDARD LVCMOS33 } [get_ports { input[5] }];  
#IO_L7N_T1_D10_14 Sch=sw[5]
```

##7 segment display

```
set_property -dict { PACKAGE_PIN T10  IOSTANDARD LVCMOS33 } [get_ports { cathodes_out[0] }];  
#IO_L24N_T3_A00_D16_14 Sch=ca
```

```
set_property -dict { PACKAGE_PIN R10  IOSTANDARD LVCMOS33 } [get_ports { cathodes_out[1] }];  
#IO_25_14 Sch=cb
```

```
set_property -dict { PACKAGE_PIN K16  IOSTANDARD LVCMOS33 } [get_ports { cathodes_out[2] }];  
#IO_25_15 Sch=cc
```

```
set_property -dict { PACKAGE_PIN K13  IOSTANDARD LVCMOS33 } [get_ports { cathodes_out[3] }];  
#IO_L17P_T2_A26_15 Sch=cd
```

```
set_property -dict { PACKAGE_PIN P15  IOSTANDARD LVCMOS33 } [get_ports { cathodes_out[4] }];  
#IO_L13P_T2_MRCC_14 Sch=ce
```

```
set_property -dict { PACKAGE_PIN T11  IOSTANDARD LVCMOS33 } [get_ports { cathodes_out[5] }];  
#IO_L19P_T3_A10_D26_14 Sch=cf
```

```
set_property -dict { PACKAGE_PIN L18  IOSTANDARD LVCMOS33 } [get_ports { cathodes_out[6] }];  
#IO_L4P_T0_D04_14 Sch=cg
```

```
set_property -dict { PACKAGE_PIN H15  IOSTANDARD LVCMOS33 } [get_ports { cathodes_out[7] }];  
#IO_L19N_T3_A21_VREF_15 Sch=dp
```

```
set_property -dict { PACKAGE_PIN J17  IOSTANDARD LVCMOS33 } [get_ports { anodes_out[0] }];  
#IO_L23P_T3_FOE_B_15 Sch=an[0]
```

```
set_property -dict { PACKAGE_PIN J18  IOSTANDARD LVCMOS33 } [get_ports { anodes_out[1] }];  
#IO_L23N_T3_FWE_B_15 Sch=an[1]
```

```
set_property -dict { PACKAGE_PIN T9   IOSTANDARD LVCMOS33 } [get_ports { anodes_out[2] }];  
#IO_L24P_T3_A01_D17_14 Sch=an[2]
```

```
set_property -dict { PACKAGE_PIN J14  IOSTANDARD LVCMOS33 } [get_ports { anodes_out[3] }];  
#IO_L19P_T3_A22_15 Sch=an[3]
```

```
set_property -dict { PACKAGE_PIN P14  IOSTANDARD LVCMOS33 } [get_ports { anodes_out[4] }];  
#IO_L8N_T1_D12_14 Sch=an[4]
```

```
set_property -dict { PACKAGE_PIN T14  IOSTANDARD LVCMOS33 } [get_ports { anodes_out[5] }];  
#IO_L14P_T2_SRCC_14 Sch=an[5]
```

```
set_property -dict { PACKAGE_PIN K2   IOSTANDARD LVCMOS33 } [get_ports { anodes_out[6] }];  
#IO_L23P_T3_35 Sch=an[6]
```

```
set_property -dict { PACKAGE_PIN U13  IOSTANDARD LVCMOS33 } [get_ports { anodes_out[7] }];  
#IO_L23N_T3_A02_D18_14 Sch=an[7]
```

##Buttons

```
set_property -dict { PACKAGE_PIN N17  IOSTANDARD LVCMOS33 } [get_ports { view }];  
#IO_L9P_T1_DQS_14 Sch=btnc
```

```
set_property -dict { PACKAGE_PIN M18  IOSTANDARD LVCMOS33 } [get_ports { input_count }];  
#IO_L4N_T0_D05_14 Sch=btneu
```

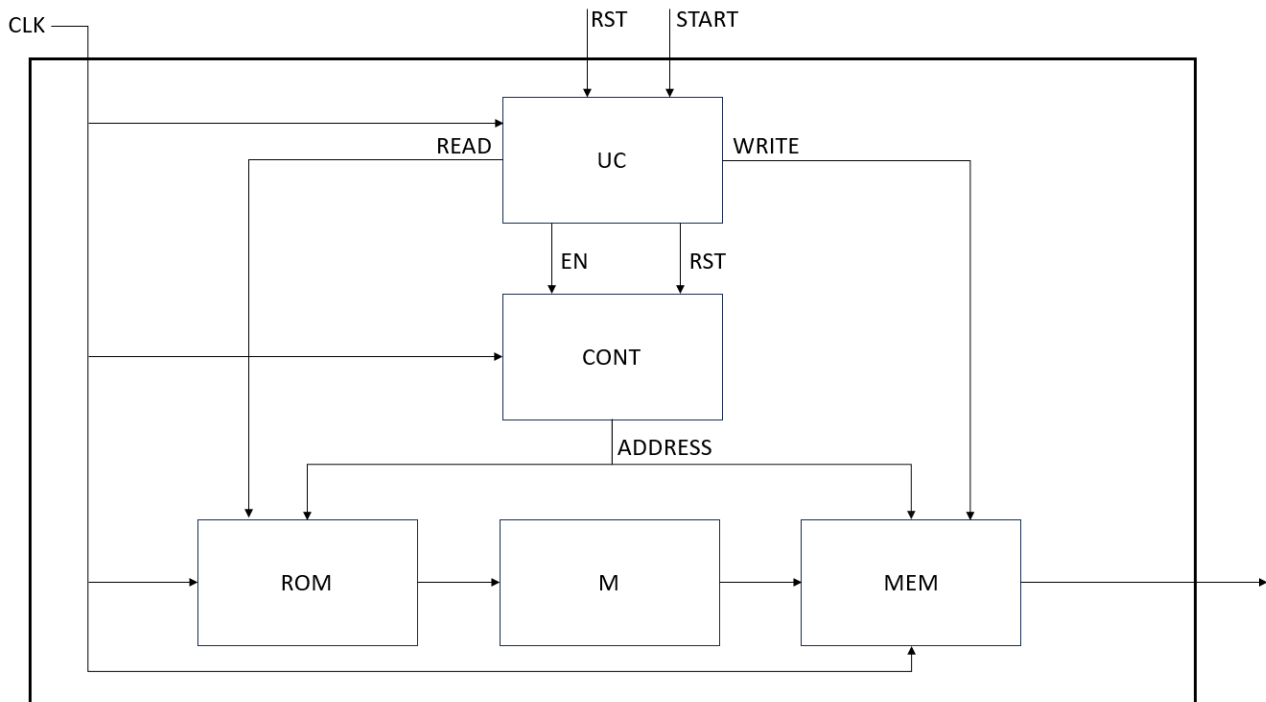
```
set_property -dict { PACKAGE_PIN P17  IOSTANDARD LVCMOS33 } [get_ports { reset }];  
#IO_L12P_T1_MRCC_14 Sch=btnd
```

```
set_property -dict { PACKAGE_PIN P18  IOSTANDARD LVCMOS33 } [get_ports { stop }];  
#IO_L9N_T1_DQS_D13_14 Sch=btnd
```

## Esercizio 6: Sistema di lettura-elaborazione-scrittura PO\_PC

### Progetto e architettura

Il sistema richiesto può essere così schematizzato nel seguente modo.



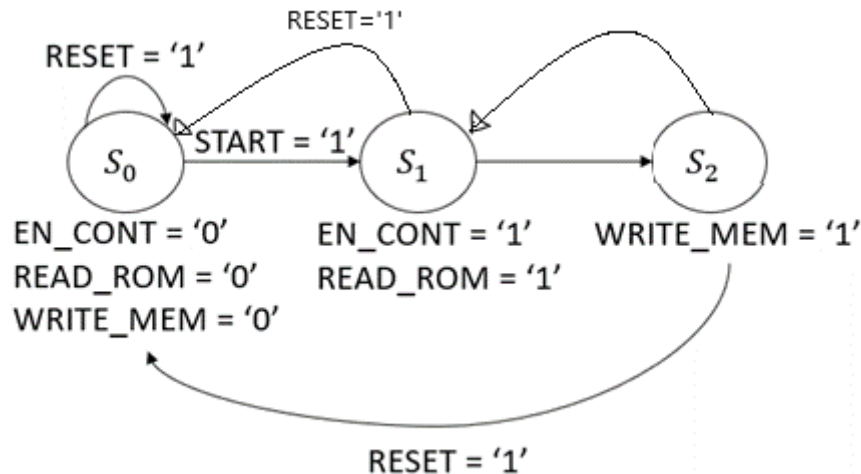
Le componenti dell'architettura rappresentata, nel dettaglio, sono le seguenti:

- **UC:** unità di controllo il cui ruolo principale è quello di fornire i segnali di controllo per il contatore e le memorie
- **ROM:** memoria di sola lettura da cui vengono prelevati i dati di partenza
- **M:** macchina combinatoria che restituisce in uscita il segnale in ingresso ma negato
- **MEM:** memoria in cui vengono immagazzinati i dati finali
- **CONT:** contatore che scandisce le locazioni delle memorie da cui leggere e in cui scrivere

Il modulo **CU** implementa una macchina a stati finiti (FSM) con tre stati (**S0**, **S1**, **S2**) che gestisce il comportamento degli altri componenti. L'entity CU ha tre porte di input (**start**, **reset**, **clk**) e una porta di uscita (**output**), che è un vettore di 4 bit.

All'interno dell'architettura sono dichiarati e istanziati componenti di tipo ROM, M, MEM e cont\_mod\_16. Questi componenti sono collegati in modo appropriato alle porte di input/output di CU e sono interconnessi fra loro attraverso l'utilizzo di segnali interni appositamente creati, ovvero **read\_ROM**, **en\_cont**, **write\_MEM**, **address**, **ROM\_out** e **M\_out**.

Il file "CU.vhd", quindi, rappresenta il sistema complessivo dato che contiene sia i due processi necessari a implementare il diagramma di stati dell'automa (descritto successivamente) sia tutte le istanze delle entità che sono controllate da questa macchina.



- Stato  $S_0$ : Questo è lo stato iniziale in cui si disabilita il contatore, la lettura dalla ROM e la scrittura nella MEM. Si può tornare in questo stato a partire da un qualsiasi altro con un reset e si rimanere in esso finché non si riceve un segnale di start.
- Stato  $S_1$ : In questo stato, la macchina abilita il contatore e la lettura dalla ROM.
- Stato  $S_2$ : In questo stato si abilita la scrittura nella MEM e poi si fa la transizione verso S1.

Il file "cont\_mod\_16.vhd" definisce un contatore modulo 16, caratterizzato da un segnale di clock (**clk**), un segnale di reset asincrono (**rst**), un segnale di abilitazione (**enable**), e produce un risultato a 4 bit denominato "**count**" che va in ingresso alle due memorie come "**address**".

Il file "ROM.vhd" rappresenta una memoria di sola lettura (ROM) caratterizzata da una struttura di tipo array a 16 posizioni, ciascuna contenente un vettore di 8 bit. I terminali di ingresso sono il segnale di sincronismo (**clk**), il segnale di lettura (**read**), l'indirizzo in cui leggere (**address**); il segnale di uscita è il vettore a 8 bit dei dati letti (**d\_out**). La memoria in questione contiene valori binari che spaziano da "00000000" in posizione "0000" a "00001111" in posizione "1111".

Il modulo M rappresenta una macchina puramente combinatoria. Quando essa riceve in ingresso un vettore di 8 bit (nell'architettura l'input è collegato all'uscita della ROM), produce immediatamente, a meno di ritardi di propagazione, un vettore di uscita di 4 bit. L'output corrisponde al negato dei 4 bit meno significativi del vettore in input (nell'architettura l'uscita è connessa all'ingresso della memoria MEM).

La MEM funge da banco di memoria a 16 posizioni, ciascuna contenente un vettore di 4 bit. L'entity presenta cinque porte: in ingresso si ha il segnale di temporizzazione (**clk**), l'indirizzo di memoria (**address**), il dato di 4 bit da scrivere (**d\_in**) e il segnale di scrittura (**write**); in uscita si ha l'ultimo dato scritto (**d\_out**).

### Implementazione

#### CU.vhd

```

library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity CU is port (
    start : in std_logic;

```

```

    reset: in std_logic;
    clk : in std_logic;
    output : out std_logic_vector(3 downto 0)
);
end CU;

```

architecture Behavioral of CU is

```

component ROM is
port (
    clk      : in  std_logic;
    read     : in  std_logic;
    address  : in  std_logic_vector(3 downto 0);
    d_out    : out std_logic_vector(7 downto 0)
);
end component;

```

```

component M is port(
    x : in  std_logic_vector(7 downto 0);
    y : out std_logic_vector(3 downto 0)
);
end component;

```

```

component MEM is
port (
    clk      : in  std_logic;
    address  : in  std_logic_vector(3 downto 0);
    d_in     : in  std_logic_vector(3 downto 0);
    write    : in  std_logic;
    d_out    : out std_logic_vector(3 downto 0)
);
end component;

```

```

component cont_mod_16
port (
    clk      : in  std_logic;
    rst      : in  std_logic;
    enable   : in  std_logic;
    count    : out std_logic_vector(3 downto 0)
);
end component;

```

```

signal read_ROM : std_logic := '0';
signal en_cont  : std_logic := '0';
signal write_MEM : std_logic := '0';
signal address  : std_logic_vector(3 downto 0) := (others=>'0');
signal ROM_out  : std_logic_vector(7 downto 0) := (others=>'0');
signal M_out    : std_logic_vector(3 downto 0) := (others=>'0');

```

```

type stato is (S0,S1,S2);

```

```
signal stato_corrente : stato := S0;  
signal stato_prossimo : stato;
```

```
begin
```

```
  mem_rom: ROM port map(  
    clk => clk,  
    read => read_ROM,  
    address => address,  
    d_out => ROM_out  
  );
```

```
  machine_m: M port map(  
    x => ROM_out,  
    y => M_out  
  );
```

```
  mem_mem: MEM port map(  
    clk => clk,  
    address => address,  
    d_in => M_out,  
    write => write_MEM,  
    d_out => output  
  );
```

```
  contatore: cont_mod_16  
    port map (  
      clk => clk,  
      rst => reset,  
      enable => en_cont,  
      count => address);
```

```
  process_1 : process(stato_corrente,start,reset)
```

```
  begin
```

```
    case stato_corrente is
```

```
      when S0 =>  
        en_cont <= '0';  
        read_ROM <= '0';  
        write_MEM <= '0';  
        if reset='1' then  
          stato_prossimo <= S0;  
        elsif(start='1') then  
          stato_prossimo <= S1;  
        else stato_prossimo <= S0;  
        end if;
```

```
      when S1 =>  
        if(reset='1') then --  
          stato_prossimo <= S0;
```

```

        else en_cont <= '1';
        read_ROM <= '1';
        stato_prossimo <= S2;
    end if;

    when S2 =>
        write_MEM <= '1';
        if(reset='1') then
            stato_prossimo <= S0;
        else stato_prossimo <= S1;
        end if;
    end case;
end process;

process_2 : process(clk)
begin
    if rising_edge(clk) then
        stato_corrente <= stato_prossimo;
    end if;
end process;
end Behavioral;

```

---

## cont\_mod\_16.vhd

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity cont_mod_16 is
    port (
        clk      : in  std_logic;
        rst      : in  std_logic;
        enable   : in  std_logic;
        count    : out std_logic_vector(3 downto 0)
    );
end entity cont_mod_16;

architecture Behavioral of cont_mod_16 is

    signal counter_value : unsigned(3 downto 0) := (others => '0');

begin

    process (clk, rst)
    begin
        if rst = '1' then
            counter_value <= (others => '0');
        elsif rising_edge(clk) and enable = '1' then

```

```

        if counter_value = 15 then
            counter_value <= (others => '0');
        else
            counter_value <= counter_value + 1;
        end if;
    end if;
end process;

count <= std_logic_vector(counter_value);

end architecture Behavioral;

```

---

## ROM.vhd

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ROM is
    port (
        clk      : in  std_logic;
        read      : in  std_logic;
        address   : in  std_logic_vector(3 downto 0);
        d_out     : out std_logic_vector(7 downto 0)
    );
end entity ROM;

architecture RTL of ROM is
    type MEMORY_16_8 is array (0 to 15) of std_logic_vector(7 downto 0);

    constant ROM_16_8 : MEMORY_16_8 := (
        "00000000",
        "00000001",
        "00000010",
        "00000011",
        "00000100",
        "00000101",
        "00000110",
        "00000111",
        "00001000",
        "00001001",
        "00001010",
        "00001011",
        "00001100",
        "00001101",
        "00001110",
        "00001111"
    );
end architecture;

```



```

        signal data : std_logic_vector(7 downto 0) := (others => '0');

begin
    process (clk)
    begin
        if rising_edge(clk) then
            if read='1' then
                data <= ROM_16_8(to_integer(unsigned(address)));
            end if;
        end if;
    end process;

    d_out <= data;

end architecture RTL;

```

---

## M.vhd

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity M is port(
    x : in  std_logic_vector(7 downto 0);
    y : out std_logic_vector(3 downto 0)
);
end entity M;

architecture Behavioral of M is
begin
    process(x)
    begin
        y <= (others => '0');
        y(3 downto 0) <= not(x(3 downto 0));
    end process;
end architecture behavioral;

```

---

## MEM.vhd

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity MEM is
    port (
        clk      : in  std_logic;
        address   : in  std_logic_vector(3 downto 0);
        d_in      : in  std_logic_vector(3 downto 0);
        write     : in  std_logic;

```

```

        d_out    : out std_logic_vector(3 downto 0)
    );
end entity MEM;

architecture RTL of MEM is
    type MEMORY_16_4 is array (0 to 15) of std_logic_vector(3 downto 0);
    signal mem_data : MEMORY_16_4 := (others => (others => '0'));

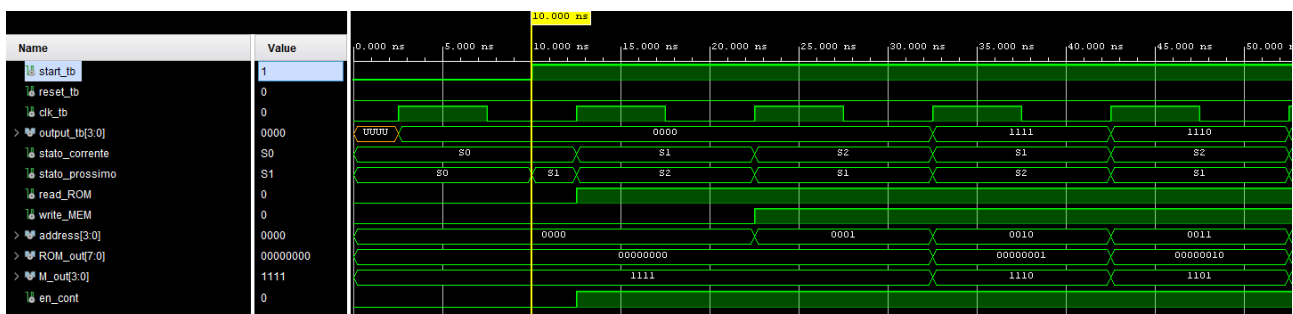
begin
    process (clk)
    begin
        if rising_edge(clk) then
            if write = '1' then
                mem_data(to_integer(unsigned(address))) <= d_in;
                d_out <= d_in;
            else
                d_out <= "0000";
            end if;
        end if;
    end process;

end architecture RTL;

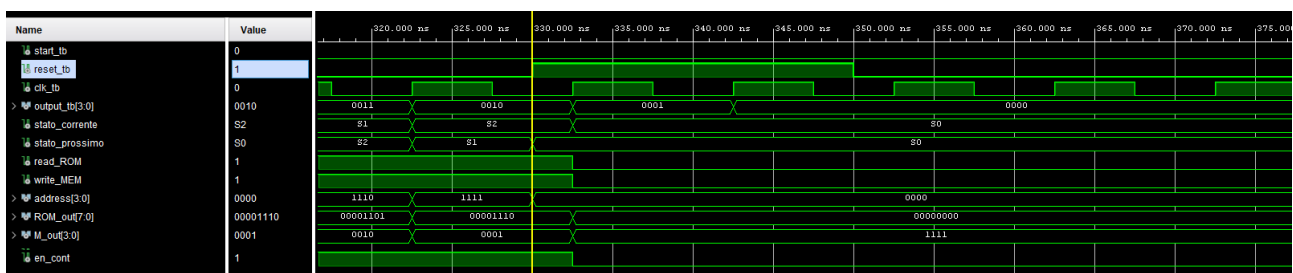
```

### Simulazione

Nella prima fase, al colpo di clock successivo all'alzarsi del segnale di start, vengono abilitati il contatore e la ROM. La macchina combinatoria M calcola il valore di uscita, invertendo il suo ingresso, e successivamente il dato ricavato viene scritto in memoria MEM abilitata tramite il segnale WRITE\_MEM (0→1).



Nella seconda fase invece, viene posto ad 1 il segnale di RESET. In questo caso, tutti i valori vengono portati a 0 e vengono "spente" le memorie e il contatore.



---

## CU\_tb.vhd

```
library IEEE;
use ieee.std_logic_1164.all;

entity CU_tb is
end entity CU_tb;

architecture testbench of CU_tb is
    signal start_tb, reset_tb, clk_tb : std_logic := '0';
    signal output_tb : std_logic_vector(3 downto 0);

    component CU
    port (
        start  : in std_logic;
        reset  : in std_logic;
        clk     : in std_logic;
        output : out std_logic_vector(3 downto 0)
    );
end component;

begin
    clock: process
    begin
        while now < 1000 ns loop
            clk_tb <= not clk_tb after 2.5 ns;
            wait for 5 ns;
        end loop;
        wait;
    end process;

    stim : process
    begin
        wait for 10 ns;
        start_tb <= '1';
        wait for 20 ns;
        wait for 80 ns;
        start_tb <= '0';
        wait for 100 ns;

        start_tb <= '1';
        wait for 20 ns;
        start_tb <= '0';

        wait for 100 ns;
        reset_tb <= '1';
        wait for 20 ns;
        reset_tb <= '0';
```

```

        wait for 300 ns;
        start_tb <= '1';
        wait for 20 ns;
        start_tb <= '0';

        wait;
    end process;

    dut: CU
        port map (
            start => start_tb,
            reset => reset_tb,
            clk   => clk_tb,
            output => output_tb
        );

end architecture testbench;

```

---

#### *Sintesi su board di sviluppo*

Premendo il pulsante **read** associato al bottone **M18** si fa partire l'elaborazione; il risultato del sistema ovvero il dato scritto in MEM viene riportato, in forma binaria, sui primi 4 **led** ovvero quelli associati ai pin che vanno da **H17** a **N14**. Il pulsante di **reset P17** è direttamente collegato al porto di ingresso "reset" della componente CU che è stata istanziata (**c\_u**). Infine, è da notare che per facilitare la visualizzazione dei vari valori sui LED, è stato generato, attraverso un process, un nuovo segnale di clock denominato "**clock**" il cui periodo è 0.5s (mezzo secondo) a partire da quello della scheda (**clk**) che ha T=10ns. Il segnale clock è quello che viene messo input a c\_u.

---

#### **CU\_on\_board.vhd**

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CU_on_board is Port (
    clk : in std_logic;
    read : in std_logic;
    reset : in std_logic;
    led : out std_logic_vector(3 downto 0));
end CU_on_board;

architecture Behavioral of CU_on_board is

    component CU is port (
        start : in std_logic;
        reset: in std_logic;
        clk : in std_logic;

```

```

        output : out std_logic_vector(3 downto 0)
    );
end component;

signal counter : integer := 0;
signal clock : std_logic := '0'; -- mezzo secondo
signal out_put : std_logic_vector(3 downto 0):=(others=>'0');

begin

    c_u : CU port map(
        start => read,
        reset => reset,
        clk => clock,
        output => out_put);

    led <= out_put;

    process(clk)
    begin
        if rising_edge(clk) then
            counter <= counter + 1;
            if counter = (50_000_000)/2 then
                counter <= 0;
                clock <= not clock;
            end if;
        end if;
    end process;

end Behavioral;

```

---

## Clock signal

```

set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports { clk }];
#IO_L12P_T1_MRCC_35 Sch=clk100mhz

```

```

create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk}];

```

## LEDs

```

set_property -dict { PACKAGE_PIN H17     IOSTANDARD LVCMOS33 } [get_ports { led[0] }];
#IO_L18P_T2_A24_15 Sch=led[0]

```

```

set_property -dict { PACKAGE_PIN K15     IOSTANDARD LVCMOS33 } [get_ports { led[1] }];
#IO_L24P_T3_RS1_15 Sch=led[1]

```

```

set_property -dict { PACKAGE_PIN J13     IOSTANDARD LVCMOS33 } [get_ports { led[2] }];
#IO_L17N_T2_A25_15 Sch=led[2]

```

```

set_property -dict { PACKAGE_PIN N14     IOSTANDARD LVCMOS33 } [get_ports { led[3] }];
#IO_L8P_T1_D11_14 Sch=led[3]

```

### ##Buttons

```
set_property -dict { PACKAGE_PIN M18      IOSTANDARD LVCMOS33 } [get_ports { read }];  
#IO_L4N_T0_D05_14 Sch=btneu  
  
set_property -dict { PACKAGE_PIN P17      IOSTANDARD LVCMOS33 } [get_ports { reset }];  
#IO_L12P_T1_MRCC_14 Sch=btneu
```

### Timing analysis

La timing analysis di un circuito digitale permette di valutare se il progetto rispetta i vincoli temporali richiesti per garantire un corretto funzionamento del sistema. I constraint temporali sono specificati sotto forma di comandi **TCL** (Tool Command Language) all'interno di un file di tipo .XDC (Xilinx Design Constraints).

Per stabilire vincoli sull'unico clock utilizzato in questo design sono stati scritti i seguenti comandi.

---

### ## Clock signal

```
set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports { clk }];  
#IO_L12P_T1_MRCC_35 Sch=clk100mhz  
  
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk}];
```

---

In questo caso, si è creato un "generated clock" (non virtuale) denominato **clk**, collegato al pin E3 della FPGA, il cui periodo (**-period**) è 10 nanosecondi e il duty cycle è del 50% (**-waveform**). Nel dettaglio "**-waveform**" consente di specificare gli istanti in ns in cui si verificano i fronti di salita e discesa all'interno di un singolo periodo del clock.

Questo segnale di sincronismo è proprio quello che viene messo in ingresso alle componenti cont\_mod\_16, CU, MEM e ROM quando si esegue la sintesi sul board tramite CU\_on\_board.

Dato questo vincolo, è possibile stimare approssimativamente la frequenza di funzionamento attuale a partire dal periodo del clock (T) e dal valore del **WNS** (Worst Negative Slack). Col termine "slack" ci si riferisce al margine temporale disponibile tra il ritardo richiesto e il ritardo effettivo in percorso all'interno del circuito. Il Worst Negative Slack rappresenta il valore minimo di questa misura nei percorsi critici del design. In altre parole, indica il delay massimo entro il quale il cammino critico deve essere "completato" affinché il sistema soddisfi i requisiti di temporizzazione; quindi, quando il WNS è negativo si potrebbero osservare comportamenti incorretti.

Il WNS può essere ottenuto consultando l'interfaccia "Design Timing Summary" nella sezione "Implementation" dell'applicazione Vivado.

#### Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 6,179 ns	Worst Hold Slack (WHS): 0,245 ns	Worst Pulse Width Slack (WPWS): 4,500 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 65	Total Number of Endpoints: 65	Total Number of Endpoints: 34
All user specified timing constraints are met.		

In questo momento il WNS è uguale a 6.179 ns; la frequenza di funzionamento del circuito è ricavata dall'espressione  $\frac{1}{T-WNS} = \frac{1}{10-6.179} \cong 0.262$ . Per determinare la frequenza massima, si riduce gradualmente il periodo fino a quando il Worst Negative Slack (WNS) rimane positivo.

```
create_clock -add -name sys_clk_pin -period 06.00 -waveform {0 3} [get_ports {clk}];
```

Design Timing Summary			
Setup	Hold		Pulse Width
Worst Negative Slack (WNS):	2,422 ns	Worst Hold Slack (WHS):	0,245 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	65	Total Number of Endpoints:	65
All user specified timing constraints are met.			

```
create_clock -add -name sys_clk_pin -period 04.00 -waveform {0 2} [get_ports {clk}];
```

Design Timing Summary			
Setup	Hold		Pulse Width
Worst Negative Slack (WNS):	0,382 ns	Worst Hold Slack (WHS):	0,280 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	65	Total Number of Endpoints:	65
All user specified timing constraints are met.			

```
create_clock -add -name sys_clk_pin -period 03.00 -waveform {0 1.5} [get_ports {clk}];
```

Design Timing Summary			
Setup	Hold		Pulse Width
Worst Negative Slack (WNS):	-0,342 ns	Worst Hold Slack (WHS):	0,275 ns
Total Negative Slack (TNS):	-7,135 ns	Total Hold Slack (THS):	0,000 ns
Number of Failing Endpoints:	45	Number of Failing Endpoints:	0
Total Number of Endpoints:	65	Total Number of Endpoints:	65
Timing constraints are not met.			

```
create_clock -add -name sys_clk_pin -period 03.50 -waveform {0 1.75} [get_ports {clk}];
```

#### Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0,044 ns	Worst Hold Slack (WHS): 0,275 ns	Worst Pulse Width Slack (WPWS): 1,250 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 65	Total Number of Endpoints: 65	Total Number of Endpoints: 34
All user specified timing constraints are met.		

La massima frequenza di funzionamento del sistema digitale in questione è quindi  $F_{MAX} = \frac{1}{T - WNS} = \frac{1}{3.5 - 0.044} \cong 0.289$



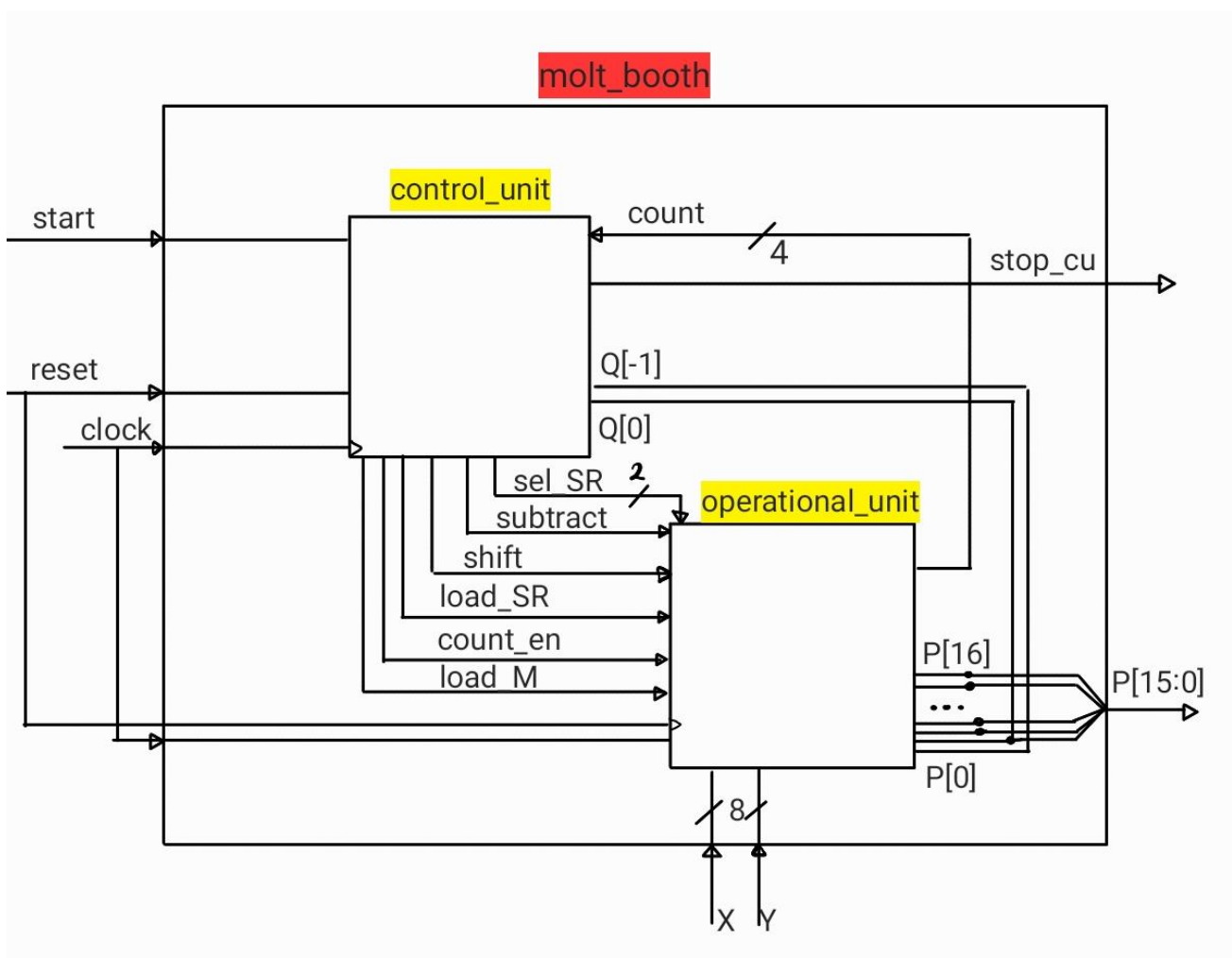
## Capitolo 3: Macchine Aritmetiche

### Esercizio 7: Moltiplicatore di Booth

#### Progetto e architettura

Il moltiplicatore di Booth (**molt\_booth**) realizza il prodotto tra due operandi con segno attraverso la composizione di due entità principali: **operational\_unit** e **control\_unit**. La macchina è modellata come un automa a stati finiti: l'unità di controllo (**control\_unit**) realizza il diagramma a stati che, attraverso la definizione di segnali di controllo, guida il flusso di esecuzione della parte operativa (**operational\_unit**), la quale svolge le operazioni necessarie per ottenere il risultato della moltiplicazione.

Di seguito è presentata l'architettura complessiva del sistema **molt\_booth**.



L'entità **control\_unit** è caratterizzata da un'architettura comportamentale che implementa l'algoritmo di Booth per la moltiplicazione di due numeri interi con segno. L'algoritmo, basato sulla codifica di Booth, è riportato successivamente e consente l'implementazione di un moltiplicatore seriale piuttosto che parallelo.

---

BoothMultiplier: (in : INBUS; out : OUTBUS)

register A[7:0], M[7:0], Q[7:-1], COUNT[2:0];

```

bus INBUS[7:0], OUTBUS[7:0];

Begin:      A := 0; COUNT := 0;
Input:      M := INBUS; Q[7:0] := INBUS; Q[-1] := 0;
Scan:       if Q[0]Q[-1] = 01
             then A[7:0] := A[7:0] + M[7:0];
             else if Q[0]Q[-1] = 10
             then A[7:0] := A[7:0] - M[7:0];
RightShift: A[7] := A[7]; A[6:0].Q := A.Q[7:0];
Increment:  COUNT := COUNT + 1; goto Scan;
Test:       if COUNT < 8 then go to Scan;
Output:     OUTBUS := A;
            OUTBUS := Q[7:0];

End BoothMultiplier;

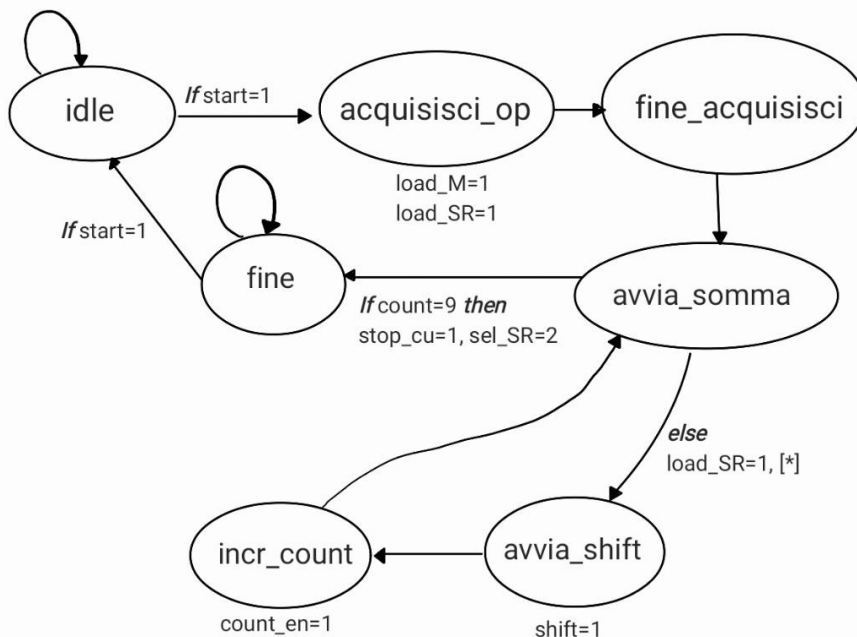
```

---

La soluzione adottata prevede la definizione di due processi.

1. **res\_stato**: Questo processo si attiva solo in corrispondenza di eventi del clock e, se si è osservato proprio un fronte di salita, imposta lo stato corrente (**current\_state**) a quello di riposo (**idle**) se il segnale di **reset** è alto, altrimenti aggiorna **current\_state** al valore dello stato successivo (**next\_state**).
2. **Comb**: Questo process include nella sensitivity list lo stato corrente (**current\_state**), il segnale d'inizio (**start**) e il conteggio (**conteggio**) e, in risposta a variazioni dei valori di tali segnali, modifica opportunamente lo stato successivo (**next\_state**) e i segnali di controllo (**count\_en**, **subtract**, **sel\_SR**, **load\_SR**, **load\_M**, **stop\_cu** e **shift**). Si osservi che ogni volta che il processo in questione si "risveglia" vengono azzerati i segnali di **count\_en**, **subtract**, **sel\_SR**, **load\_SR**, **load\_M**, **stop\_cu** e **shift** per essere poi eventualmente alzati se lo stato lo prevede.

Il diagramma degli stati finiti qui di seguito mostra in dettaglio tutti i possibili stati in cui la macchina può trovarsi, le transizioni tra gli stati in risposta agli eventi, e le azioni svolte in ciascuno di essi.



[\*]  
 If  $Q[0]=0$  and  $Q[-1]=1$  then  
 sel\_SR=1, subtract=0  
  
 If  $Q[0]=1$  and  $Q[-1]=0$  then  
 sel\_SR=1, subtract=1  
  
 If  $(Q[0]=0$  and  $Q[-1]=0)$  or  $(Q[0]=1$   
 and  $Q[-1]=1)$  then sel\_SR=2

L'entity operational\_unit è invece realizzata attraverso un approccio strutturale. Le componenti istanziate sono uno shift register (**shift\_register**) a 17 bit, rappresentante la concatenazione delle parole A da 8 bit e Q da 9 bit, un registro (**M**) da 8 bit per contenere l'operando Y, un contatore modulo 9 (**counter**), un sommatore/sottrattore (**parallel\_adder**) di due interi a 8 bit basato sul ripple carry adder, e infine un multiplexer 4 a 1 (**mux\_41**) per selezionare la word da scrivere parallelamente nello shift register.

- L'entità shift\_register ha cinque porte: **parallel\_in** per il caricamento parallelo dei dati, **clock** per il segnale di clock, **reset** per il reset, **load** per il comando di inserimento dei dati, **shift** per eseguire uno shift verso destra di una posizione, e **parallel\_out** per la lettura parallela dei dati.  
 L'architettura "Behavioural" include un processo che gestisce il comportamento del registro a scorrimento. Quando il segnale di clock è in salita (**clock'event and clock='1'**), il processo esegue delle operazioni in base ai segnali di reset, load e shift: se il reset è attivo, il registro viene azzerato; se il segnale di load è attivo, il registro viene caricato con i dati presenti su parallel\_in; se, invece, il segnale di shift è alto, viene fatto uno shift a destra di un bit dei dati nel registro.
- L'entità M presenta i terminali **Y\_in** per l'input del moltiplicando, **clk** per il segnale di clock, **rst** per il segnale di reset, **load** per il caricamento dei dati, e **Y\_out** per l'output del moltiplicando.  
 L'architettura "Behavioural" implementa un processo nel quale, quando si ha un fronte di salita del segnale di clock, se il reset è attivo, il registro viene azzerato; invece, se è il segnale di load ad essere uno, sull'uscita Y\_out vengono caricati i dati presenti sull'ingresso Y\_in.
- L'entità counter ha i ports: **clock** per il segnale di clock, **reset** per il segnale di reset, **enable** per abilitare il contatore, e **count** per l'uscita a 4 bit, ossia il conteggio.  
 Il contatore comportamentale è sincronizzato sulla transizione da 0 a 1 del segnale di clock: se il segnale di reset è alto, il contatore viene azzerato; in caso contrario, se il segnale di abilitazione è attivo, il contatore viene incrementato di uno. Quando il contatore raggiunge il valore massimo 9 viene riportato a 0.
- L'entità parallel\_adder ha i terminali **X** e **Y** come data input, **cin** come input per il bit di carry in (riporto in ingresso), **Z** come output per il risultato della somma/sottrazione, e **cout** come output per il bit di carry out (riporto in uscita).  
 L'architettura "Structural" implementa il sommatore/sottrattore utilizzando un componente interno di tipo **ripple\_carry**. Questo componente è un sommatore a 8 bit che riceve come operandi **X** e

Si noti che nel sommatore di tipo ripple carry il secondo operando viene calcolato mediante il costrutto **for...generate** e che il modulo è a sua volta realizzato per composizione di 8 **full\_adder**.

- L'architettura comportamentale contiene un processo sensibile ai segnali:  $s$ ,  $x_0$ ,  $x_1$ ,  $x_2$  e  $x_3$ . La struttura del multiplexer è realizzata tramite l'uso della clausola **case** la quale assegna all'uscita  $y$  uno degli ingressi in base al valore di  $s$ . I casi sono esaustivi ovvero coprono tutte le possibili combinazioni del segnale selettore.

The diagram illustrates the internal structure of the **operational\_unit**. It contains the following components and connections:

- shift\_register**: Receives **load\_SR**, **clock**, **reset**, and **shift** signals. It outputs **SR\_out** (17 bits) to the **M** block and **SR\_out[16:9]** (8 bits) to the **parallel\_adder**.
- mux\_41**: A 4-to-1 multiplexer with inputs **SR\_init**, **SR\_sum**, **SR\_out**, and a constant  $\emptyset$ . It is selected by **sel\_SR** (2 bits) and outputs **SR\_in** (17 bits) to the **shift\_register**.
- parallel\_adder**: Takes **SR\_out[16:9]** and **subtract** (8 bits) as inputs. It outputs **sum**, **c\_out**, and **c in** (8 bits) to the **counter**.
- M**: A block that receives **reset** and **clock** signals and outputs **Y** (8 bits) to the **counter**.
- counter**: Receives **clock**, **reset**, **count\_en**, and **c in** signals. It outputs **count**.

At the bottom, a code box defines the initialization and summation logic:

```

SR_init = "00000000" & X & "0"
SR_sum = sum & SR_out[8:0]
  
```

### Implementazione

Qui sono riportati i file in formato VHDL delle principali componenti del sistema.

---

#### molt\_booth.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity molt_booth is
    port( clock, reset, start: in std_logic;
          X, Y: in std_logic_vector(7 downto 0);
          P: out std_logic_vector(15 downto 0);
          stop_cu: out std_logic);
end molt_booth;

architecture Structural of molt_booth is

    component control_unit is
        port( q0, q_1, clock, reset, start: in std_logic;
              count: in std_logic_vector(3 downto 0);
              load_M, count_en, load_SR, shift: out std_logic;
              subtract, stop_cu: out std_logic;
              sel_SR : out std_logic_vector(1 downto 0));
    end component;

    component operational_unit is
        port( X, Y: in std_logic_vector(7 downto 0);
              clock, reset: in std_logic;
              load_SR, shift, load_M, subtract, count_en: in std_logic;
              sel_SR : in std_logic_vector(1 downto 0);
              count: out std_logic_vector(3 downto 0);
              P: out std_logic_vector(16 downto 0));
    end component;

    signal temp_q0,temp_q_1  : std_logic := '0';
    signal temp_sel_M, temp_subtract, temp_load_SR: std_logic;
    signal temp_sel_SR: std_logic_vector(1 downto 0);
    signal temp_count: std_logic_vector(3 downto 0);
    signal temp_p: std_logic_vector(16 downto 0);
    signal temp_count_en: std_logic;
    signal temp_shift: std_logic;
    signal temp_load_M: std_logic;

begin

    unita_di_controllo: control_unit port map
        (q0 => temp_q0,
         q_1 => temp_q_1,
```

```

        clock => clock,
        reset => reset,
        start => start,
        count => temp_count,
        load_M => temp_load_M,
        count_en => temp_count_en,
        load_SR => temp_load_SR,
        shift => temp_shift,
        subtract => temp_subtract,
        stop_cu => stop_cu,
        sel_SR => temp_sel_SR);

unita_operativa: operational_unit port map
    (X => X,
     Y => Y,
     clock => clock,
     reset => reset,
     load_SR => temp_load_SR,
     shift => temp_shift,
     load_M => temp_load_M,
     subtract => temp_subtract,
     count_en => temp_count_en,
     sel_SR => temp_sel_SR,
     count => temp_count,
     P => temp_p);

temp_q0 <= temp_p(1);
temp_q_1 <= temp_p(0);
P <= temp_p(16 downto 1);

end Structural;

```

---

## control\_unit.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity control_unit is
    port( q0, q_1, clock, reset, start: in std_logic;
          count: in std_logic_vector(3 downto 0);
          load_M, count_en, load_SR, shift: out std_logic;
          subtract, stop_cu: out std_logic;
          sel_SR : out std_logic_vector(1 downto 0));
end control_unit;

architecture structural of control_unit is

```

```

    type state is (idle, acquisisci_op, fine_acquisisci, avvia_somma,
avvia_shift, incr_count, fine);
    signal current_state,next_state: state;

begin

    reg_stato: process(clock)
    begin
        if(clock'event and clock='1') then
            if(reset='1') then
                current_state <=idle;
            else
                current_state <=next_state;
            end if;
        end if;
    end process;

    comb: process(current_state, start, count)
    begin
        count_en <='0';
        subtract <='0';
        sel_SR <= "00";
        load_SR <='0';
        load_M <='0';
        stop_cu <='0';
        shift <='0';

        case current_state is

            when idle =>
                if(start='1') then
                    next_state <= acquisisci_op;
                else
                    next_state <= idle;
                end if;

            when acquisisci_op =>
                load_M <='1';
                load_SR <='1';
                next_state <= fine_acquisisci;

            when fine_acquisisci =>
                next_state <= avvia_somma;

            when avvia_somma =>
                if(count="1000") then -- 9
                    stop_cu <= '1';
                    sel_SR <= "10";
                    next_state <= fine;
                elsif(q0='0' and q_1='1') then

```

```

        load_SR <= '1';
        sel_SR <= "01";
        subtract <= '0';
        next_state <= avvia_shift;
    elsif(q0='1' and q_1='0') then
        load_SR <= '1';
        sel_SR <= "01";
        subtract <= '1';
        next_state <= avvia_shift;
    else -- q0=0=q_1 oppure q0=1=q_1
        load_SR <= '1';
        sel_SR <= "10";
        next_state <= avvia_shift;
    end if;

    when avvia_shift =>
        shift <= '1';
        next_state <= incr_count;

    when incr_count =>
        count_en <= '1';
        next_state <= avvia_somma;

    when fine =>
        if(start/='1') then
            next_state <= fine;
        else next_state <= idle;
        end if;
    end case;
end process;
end structural;

```

---

## operational\_unit.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity operational_unit is
    port( X, Y: in std_logic_vector(7 downto 0); --moltiplicatore e
    moltiplicando
        clock, reset: in std_logic;
        load_SR, shift, load_M, subtract, count_en: in std_logic;
        sel_SR : in std_logic_vector(1 downto 0);
        count: out std_logic_vector(3 downto 0);
        P: out std_logic_vector(16 downto 0));
end operational_unit;

architecture Structural of operational_unit is

```



```

component parallel_adder is
port( X, Y: in std_logic_vector(7 downto 0);
      cin: in std_logic;
      Z: out std_logic_vector(7 downto 0);
      cout: out std_logic);
end component;

component M is
port( Y_in: in std_logic_vector(7 downto 0);
      clk, rst, load: in std_logic;
      Y_out: out std_logic_vector(7 downto 0));
end component;

component shift_register is
port( parallel_in: in std_logic_vector(16 downto 0);
      clock, reset, load, shift: in std_logic;
      parallel_out: out std_logic_vector(16 downto 0));
end component;

component counter is
port( clock, reset: in std_logic;
      enable: in std_logic;
      count: out std_logic_vector(3 downto 0));
end component;

component mux_41 is
generic (width : integer range 0 to 32 := 17);
port( x0, x1,x2,x3: in std_logic_vector(width-1 downto 0);
      s: in std_logic_vector(1 downto 0);
      y: out std_logic_vector(width-1 downto 0));
end component;

signal op_2: std_logic_vector(7 downto 0);
signal SR_init: std_logic_vector(16 downto 0);
signal SR_in: std_logic_vector(16 downto 0);
signal SR_out: std_logic_vector(16 downto 0);
signal sum: std_logic_vector(7 downto 0);
signal SR_sum : std_logic_vector(16 downto 0);
signal carry_out: std_logic; -- non utilizzato

begin

multiplicando: M port map(
  Y_in => Y,
  clk => clock,
  rst => reset,
  load => load_M,
  Y_out => op_2);

```

```

SR_init <= "00000000" & X & "0";

SR_sum <= sum & SR_out(8 downto 0);

MUX_SR_parallel_in : mux_41 generic map (width => 17) port map(
    x0 => SR_init,
    x1 => SR_sum,
    x2 => SR_out,
    x3 => "00000000000000000", -- mai selezionato
    s => sel_SR,
    y => SR_in);

registro_shift: shift_register port map(
    parallel_in => SR_in,
    clock => clock,
    reset => reset,
    load => load_SR,
    shift => shift,
    parallel_out => SR_out);

sommatore_sottrattore: parallel_adder port map(
    X => SR_out(16 downto 9),
    Y => op_2,
    cin => subtract,
    Z => sum,
    cout => carry_out);

contatore_mod_9: counter port map(
    clock => clock,
    reset => reset,
    enable => count_en,
    count => count);

P <= SR_out;

end Structural;

```

---

## shift\_register.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity shift_register is
    port( parallel_in: in std_logic_vector(16 downto 0); -- (8 bit per A) &
        (9 bit per Q)
        clock, reset, load, shift: in std_logic;
        parallel_out: out std_logic_vector(16 downto 0));
end shift_register;

```

```

architecture Behavioural of shift_register is

    signal temp: std_logic_vector(16 downto 0);

begin

    registro_a_scorrimento: process(clock)
    begin
        if(clock'event and clock='1') then
            if(reset='1') then
                temp <= (others=>'0');
            else
                if(load='1') then
                    temp <= parallel_in;
                elsif(shift='1') then
                    temp(14 downto 0) <= temp(15 downto 1);
                end if;
            end if;
        end if;
    end process;

    parallel_out <= temp;

end Behavioural;

```

---

## M.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity M is -- registro che mantiene il moltiplicando Y
    port( Y_in: in std_logic_vector(7 downto 0);
          clk, rst, load: in std_logic;
          Y_out: out std_logic_vector(7 downto 0));
end M;

architecture Behavioural of M is

    signal y: std_logic_vector(7 downto 0);

begin

    molt: process(clk)
    begin
        if(clk'event and clk='1') then
            if(rst='1') then
                y <= (others=>'0');
            else
                if(load='1') then

```

```

        y <= Y_in;
    end if;
end if;
end if;
end process;

Y_out<=y;

end Behavioural;

```

---

## counter.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity counter is -- modulo 9
    port(
        clock, reset: in std_logic;
        enable: in std_logic;
        count: out std_logic_vector(3 downto 0)
    );
end counter;

architecture Behavioural of counter is

    signal c: std_logic_vector(3 downto 0) := (others=>'0');

begin

    contatore_mod_9: process(clock)
    begin
        if(rising_edge(clock)) then
            if(reset='1') then
                c <= (others=>'0');
            else
                if(enable = '1') then
                    if (unsigned(c) = 9) then
                        c <= (others=>'0');
                    else
                        c <= std_logic_vector(unsigned(c) + 1);
                    end if;
                end if;
            end if;
        end if;
    end process;

    count <= c;

```

```
end Behavioural;
```

---

### parallel\_adder.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity parallel_adder is -- sommatore & sottrattore
    port( X, Y: in std_logic_vector(7 downto 0);
          cin: in std_logic;
          Z: out std_logic_vector(7 downto 0);
          cout: out std_logic);
end parallel_adder;

architecture Structural of parallel_adder is

    component ripple_carry is
    port( X, Y: in std_logic_vector(7 downto 0);
          c_in: in std_logic;
          c_out: out std_logic;
          Z: out std_logic_vector(7 downto 0));
    end component;

    signal compl_y: std_logic_vector(7 downto 0);

begin

    complemento_y: for i in 0 to 7 generate
        compl_y(i) <= Y(i) xor cin;
    end generate;

    ripple_carry_adder: ripple_carry port map(
        X => X,
        Y => compl_y,
        c_in => cin,
        c_out => cout,
        Z => Z);

end structural;
```

---

### ripple\_carry.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ripple_carry is
    port( X, Y: in std_logic_vector(7 downto 0);
          c_in: in std_logic;
          c_out: out std_logic;
```

```

        Z: out std_logic_vector(7 downto 0));
end ripple_carry;

architecture Structural of ripple_carry is
    component full_adder is port(
        a, b: in std_logic;
        cin: in std_logic;
        cout, s: out std_logic);
    end component;

    signal temp: std_logic_vector(7 downto 0);

begin

    full_adder_0: full_adder port map(a=>X(0), b=>Y(0), cin=>c_in,
cout=>temp(0), s=>Z(0));

    full_adder_1_to_6: for i in 1 to 6 generate
        f_a: full_adder port map(a=>X(i), b=>Y(i), cin=>temp(i-1),
cout=>temp(i), s=>Z(i));
    end generate;

    full_adder_7: full_adder port map(a=>X(7), b=>Y(7), cin=>temp(6),
cout=>c_out, s=>Z(7));

end Structural;

```

---

### *Simulazione*

È stato creato un singolo testbench (**molt\_booth\_tb**) per verificare il corretto funzionamento dell'intero sistema di moltiplicazione.

---

### **molt\_booth\_tb.vhd**

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity molt_booth_tb is
end molt_booth_tb;

architecture Behavioral of molt_booth_tb is
    signal clock : std_logic := '0';
    signal reset : std_logic := '0';
    signal start : std_logic := '0';
    signal X, Y : std_logic_vector(7 downto 0);
    signal P : std_logic_vector(15 downto 0);

```

```

signal stop_cu : std_logic;

constant CLOCK_PERIOD : time := 10 ns;
begin
    uut: entity work.molt_booth
        port map (
            clock => clock,
            reset => reset,
            start => start,
            X => X,
            Y => Y,
            P => P,
            stop_cu => stop_cu
        );

    clk_process: process
    begin
        wait for CLOCK_PERIOD / 2;
        clock <= not clock;
    end process;

    stim: process
    begin
        start <= '0';
        X <= "00000110"; -- 6
        Y <= "11111011"; -- -5
        wait for CLOCK_PERIOD;
        start <= '1';
        wait for CLOCK_PERIOD;
        start <= '0';

        while stop_cu /= '1' loop -- aspetta che la moltiplicazione finisca
            wait for CLOCK_PERIOD;
        end loop;

        wait for 3*CLOCK_PERIOD;
        reset <= '1';
        wait for 10 ns;
        reset<='0';

        X <= "00000101"; -- 5
        Y <= "00000010"; -- 2
        start <= '1';
        wait for CLOCK_PERIOD;
        start <= '0';

        while stop_cu /= '1' loop
            wait for CLOCK_PERIOD;
        end loop;
    end process;
end;

```

```

wait for 3*CLOCK_PERIOD;
reset <= '1';
wait for 10 ns;
reset<='0';

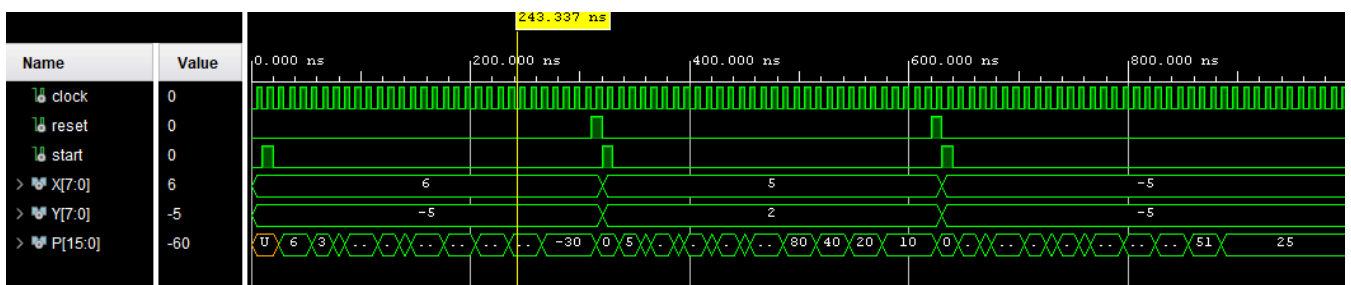
X <= "11111011"; -- -5
Y <= "11111011"; -- -5
start <= '1';
wait for CLOCK_PERIOD;
start <= '0';

while stop_cu /= '1' loop
    wait for CLOCK_PERIOD;
end loop;

wait;
end process;

end Behavioral;

```



### Sintesi su board di sviluppo

Per fare la sintesi su board di sviluppo, è necessario sviluppare un ulteriore file VHDL denominato **molt\_booth\_on\_board** all'interno del quale si istanzia il moltiplicatore di Booth, nonché il modulo **display\_seven\_segments**, il quale consente la visualizzazione del risultato in formato esadecimale sui display, e il **ButtonDebouncer** applicato al bottone di **start**. Il ButtonDebouncer è una componente cruciale che svolge la funzione di mitigare gli effetti indesiderati causati da eventuali rimbalzi del segnale di input proveniente dal pulsante di avvio.

Gli input di **molt\_booth\_on\_board** sono **clk**, ovvero il segnale di temporizzazione il cui periodo è di 10ns, **reset** collegato al bottone in posizione N17, **start** collegato al pulsante P17, **insert** collegato a P18, il vettore di 8 bit **X** associato agli switch dal pin **R13** al pin **J15** e **Y** associato agli switch da **V10** a **T8**. Gli output sono, invece, **anodes\_out** e **cathodes\_out**, i quali sono direttamente connessi agli anodi e ai catodi del display.

Premendo **insert**, vengono acquisiti, dagli switch, i valori degli operandi del prodotto (in forma binaria e con segno); successivamente, premendo **start** si avvia l'elaborazione alla fine della quale il risultato viene stampato sulle prime 4 cifre del display. Prima di iniziare una nuova moltiplicazione è necessario premere il pulsante di **reset**.



---

## molt\_booth\_on\_board.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity molt_booth_on_board is
    Port (clk : in std_logic;
          reset : in std_logic; -- da premere prima di calcolare il prossimo
prodotto
          start : in std_logic;
          insert : in std_logic;
          X : in std_logic_vector(7 downto 0);
          Y : in std_logic_vector(7 downto 0);
          anodes_out : out std_logic_vector (7 downto 0);
          cathodes_out : out std_logic_vector (7 downto 0)
          );

end molt_booth_on_board;

architecture Behavioral of molt_booth_on_board is
    component molt_booth is
        port( clock, reset, start: in std_logic;
              X, Y: in std_logic_vector(7 downto 0);
              P: out std_logic_vector(15 downto 0);
              stop_cu: out std_logic);
    end component;

    component ButtonDebouncer is
        generic (
            CLK_period: integer := 10;
            btn_noise_time: integer := 10000000
        );
        Port ( RST : in STD_LOGIC;
              CLK : in STD_LOGIC;
              BTN : in STD_LOGIC;
              CLEARED_BTN : out STD_LOGIC);
    end component;

    component display_seven_segments
        generic(
            CLKIN_freq : integer := 100000000;
            CLKOUT_freq : integer := 500
        );
        PORT(
            CLK : IN std_logic;
            RST : IN std_logic;
            VALUE : IN std_logic_vector(31 downto 0);
```

```

    ENABLE : IN std_logic_vector(7 downto 0);
    DOTS : IN std_logic_vector(7 downto 0);
    ANODES : OUT std_logic_vector(7 downto 0);
    CATHODES : OUT std_logic_vector(7 downto 0)
  );
end component;

signal x_in, y_in : std_logic_vector(7 downto 0);
signal value : std_logic_vector(31 downto 0);
signal P : std_logic_vector(15 downto 0);
signal stop_cu : std_logic;
signal start_debounced : std_logic;
begin
  process(clk)
  begin
    if(insert='1') then
      x_in <= X;
      y_in <= Y;
    end if;
  end process;

  debouncer : ButtonDebouncer generic map (CLK_period => 10, btn_noise_time
=> 10000000) port map (RST => reset, CLK => clk, BTN => start, CLEARED_BTN =>
start_debounced);

  multicatore_booth : molt_booth port map(clock => clk, reset => reset,
start => start_debounced, X => x_in, Y => y_in, P => P, stop_cu => stop_cu);

  process(clk)
  begin
    if(reset='1') then
      value <= (others=>'0');
    elsif(stop_cu='1') then
      value <= "0000000000000000" & P;
    end if;
  end process;

  seven_segment_array: display_seven_segments GENERIC MAP(
CLKIN_freq => 100000000,
CLKOUT_freq => 500
)
  PORT MAP(
    CLK => clk,
    RST => reset,
    value => value,
    enable => "00001111",
    dots => "00000000",
    anodes => anodes_out,
    cathodes => cathodes_out
  );

```

end Behavioral;

---

## Clock signal

set\_property -dict { PACKAGE\_PIN E3     IOSTANDARD LVCMOS33 } [get\_ports { clk }];  
#IO\_L12P\_T1\_MRCC\_35 Sch=clk100mhz

create\_clock -add -name sys\_clk\_pin -period 10.00 -waveform {0 5} [get\_ports {clk}];

##Switches

set\_property -dict { PACKAGE\_PIN J15    IOSTANDARD LVCMOS33 } [get\_ports { X[0] }];  
#IO\_L24N\_T3\_RS0\_15 Sch=sw[0]

set\_property -dict { PACKAGE\_PIN L16    IOSTANDARD LVCMOS33 } [get\_ports { X[1] }];  
#IO\_L3N\_T0\_DQS\_EMCCLK\_14 Sch=sw[1]

set\_property -dict { PACKAGE\_PIN M13    IOSTANDARD LVCMOS33 } [get\_ports { X[2] }];  
#IO\_L6N\_T0\_D08\_VREF\_14 Sch=sw[2]

set\_property -dict { PACKAGE\_PIN R15    IOSTANDARD LVCMOS33 } [get\_ports { X[3] }];  
#IO\_L13N\_T2\_MRCC\_14 Sch=sw[3]

set\_property -dict { PACKAGE\_PIN R17    IOSTANDARD LVCMOS33 } [get\_ports { X[4] }];  
#IO\_L12N\_T1\_MRCC\_14 Sch=sw[4]

set\_property -dict { PACKAGE\_PIN T18    IOSTANDARD LVCMOS33 } [get\_ports { X[5] }];  
#IO\_L7N\_T1\_D10\_14 Sch=sw[5]

set\_property -dict { PACKAGE\_PIN U18    IOSTANDARD LVCMOS33 } [get\_ports { X[6] }];  
#IO\_L17N\_T2\_A13\_D29\_14 Sch=sw[6]

set\_property -dict { PACKAGE\_PIN R13    IOSTANDARD LVCMOS33 } [get\_ports { X[7] }];  
#IO\_L5N\_T0\_D07\_14 Sch=sw[7]

set\_property -dict { PACKAGE\_PIN T8     IOSTANDARD LVCMOS18 } [get\_ports { Y[0] }];  
#IO\_L24N\_T3\_34 Sch=sw[8]

set\_property -dict { PACKAGE\_PIN U8     IOSTANDARD LVCMOS18 } [get\_ports { Y[1] }];  
#IO\_25\_34 Sch=sw[9]

set\_property -dict { PACKAGE\_PIN R16    IOSTANDARD LVCMOS33 } [get\_ports { Y[2] }];  
#IO\_L15P\_T2\_DQS\_RDWR\_B\_14 Sch=sw[10]

set\_property -dict { PACKAGE\_PIN T13    IOSTANDARD LVCMOS33 } [get\_ports { Y[3] }];  
#IO\_L23P\_T3\_A03\_D19\_14 Sch=sw[11]

set\_property -dict { PACKAGE\_PIN H6     IOSTANDARD LVCMOS33 } [get\_ports { Y[4] }];  
#IO\_L24P\_T3\_35 Sch=sw[12]

set\_property -dict { PACKAGE\_PIN U12    IOSTANDARD LVCMOS33 } [get\_ports { Y[5] }];  
#IO\_L20P\_T3\_A08\_D24\_14 Sch=sw[13]

set\_property -dict { PACKAGE\_PIN U11    IOSTANDARD LVCMOS33 } [get\_ports { Y[6] }];  
#IO\_L19N\_T3\_A09\_D25\_VREF\_14 Sch=sw[14]

```
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { Y[7] }];  
#IO_L21P_T3_DQS_14 Sch=sw[15]
```

##7 segment display

```
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports {  
cathodes_out[0] }]; #IO_L24N_T3_A00_D16_14 Sch=ca
```

```
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports {  
cathodes_out[1] }]; #IO_25_14 Sch=cb
```

```
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports {  
cathodes_out[2] }]; #IO_25_15 Sch=cc
```

```
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports {  
cathodes_out[3] }]; #IO_L17P_T2_A26_15 Sch=cd
```

```
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports {  
cathodes_out[4] }]; #IO_L13P_T2_MRCC_14 Sch=ce
```

```
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports {  
cathodes_out[5] }]; #IO_L19P_T3_A10_D26_14 Sch=cf
```

```
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports {  
cathodes_out[6] }]; #IO_L4P_T0_D04_14 Sch=cg
```

```
set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports {  
cathodes_out[7] }]; #IO_L19N_T3_A21_VREF_15 Sch=dp
```

```
set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { anodes_out[0]  
}]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
```

```
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { anodes_out[1]  
}]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
```

```
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { anodes_out[2]  
}]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
```

```
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { anodes_out[3]  
}]; #IO_L19P_T3_A22_15 Sch=an[3]
```

```
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { anodes_out[4]  
}]; #IO_L8N_T1_D12_14 Sch=an[4]
```

```
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { anodes_out[5]  
}]; #IO_L14P_T2_SRCC_14 Sch=an[5]
```

```
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { anodes_out[6]  
}]; #IO_L23P_T3_35 Sch=an[6]
```

```
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports {  
anodes_out[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]
```

##Buttons

```
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { reset }];  
#IO_L9P_T1_DQS_14 Sch=btnc
```

```
set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { start }];  
#IO_L12P_T1_MRCC_14 Sch=btnl
```

```
set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { insert }];  
#IO_L9N_T1_DQS_D13_14 Sch=btnd
```

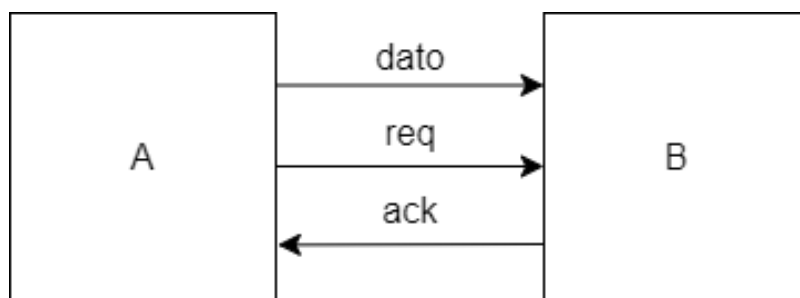
## Capitolo 4: Comunicazione con Handshaking

### Esercizio 8: Comunicazione con Handshaking

#### Progetto e architettura

Sono stati progettati due sistemi, denominati A e B, che comunicano tramite un protocollo di handshaking semplice. In particolare, A deve trasmettere dati a B; quindi, i segnali di controllo che gestiscono la comunicazione sono due; **REQ** pilotato dal sistema A e **ACK** controllato da B; e questi sono “interlacciati” tra di loro nel seguente modo.

- **REQ = 0 → 1**: il sistema A ha scritto nuovi dati sui bus dedicati (**dato**), quindi, in maniera sincrona al proprio clock (**clkA**), alza il segnale REQ per avvisare B
- **ACK = 0 → 1**: Quando il sistema B rileva REQ alto, campionando rispetto al proprio clock **clkB**, legge i dati e alza il segnale ACK per informare A dell'avvenuta ricezione.
- **REQ = 1 → 0**: Quando A vede, su un fronte di attivazione di **clkA**, che il segnale di ACK è '1' sa di poter terminare la comunicazione e abbassa REQ.
- **ACK = 1 → 0**: Quando B rileva la discesa di REQ (campionato su **clkB**) porta ACK a '0'.



Il sistema A è composto da:

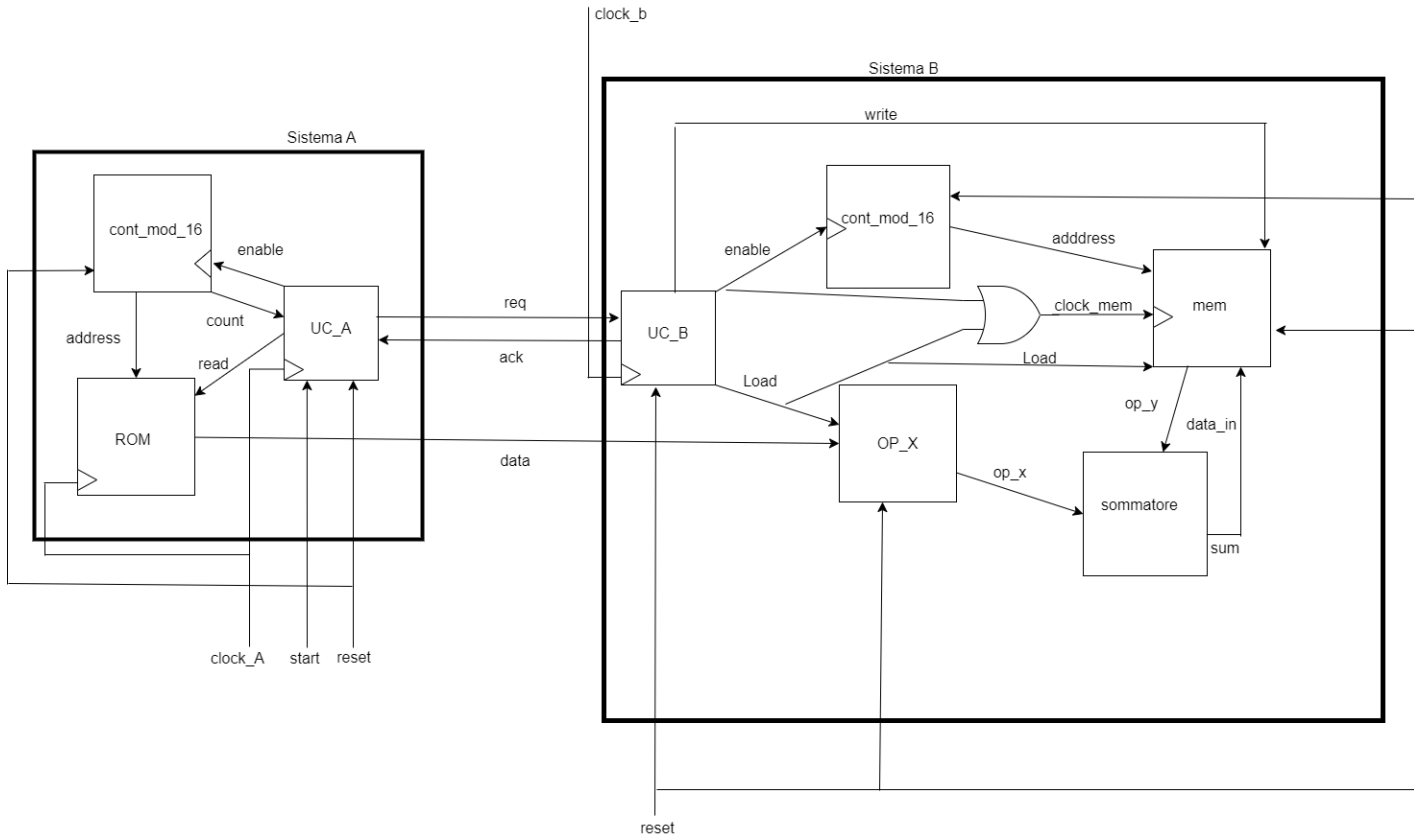
- Un'unità di controllo che comunica con il sistema B (inviando il segnale **req** e ricevendo il segnale **ack**) e fornisce agli altri componenti di A vari segnali di controllo, come il segnale **enable** per attivare il contatore e il segnale **read** alla ROM per ottenere il dato contenuto alla locazione **address**.
- Un contatore modulo 16 che riceve il segnale di abilitazione **enable** e fornisce il risultato del conteggio alla ROM, rappresentante l'indirizzo in cui leggere.
- Una ROM che riceve in ingresso una posizione e restituisce in uscita il dato attualmente presente in quel punto della memoria. Queste ultime informazioni sono quelle trasmesse a B.

Il sistema B è composto da:

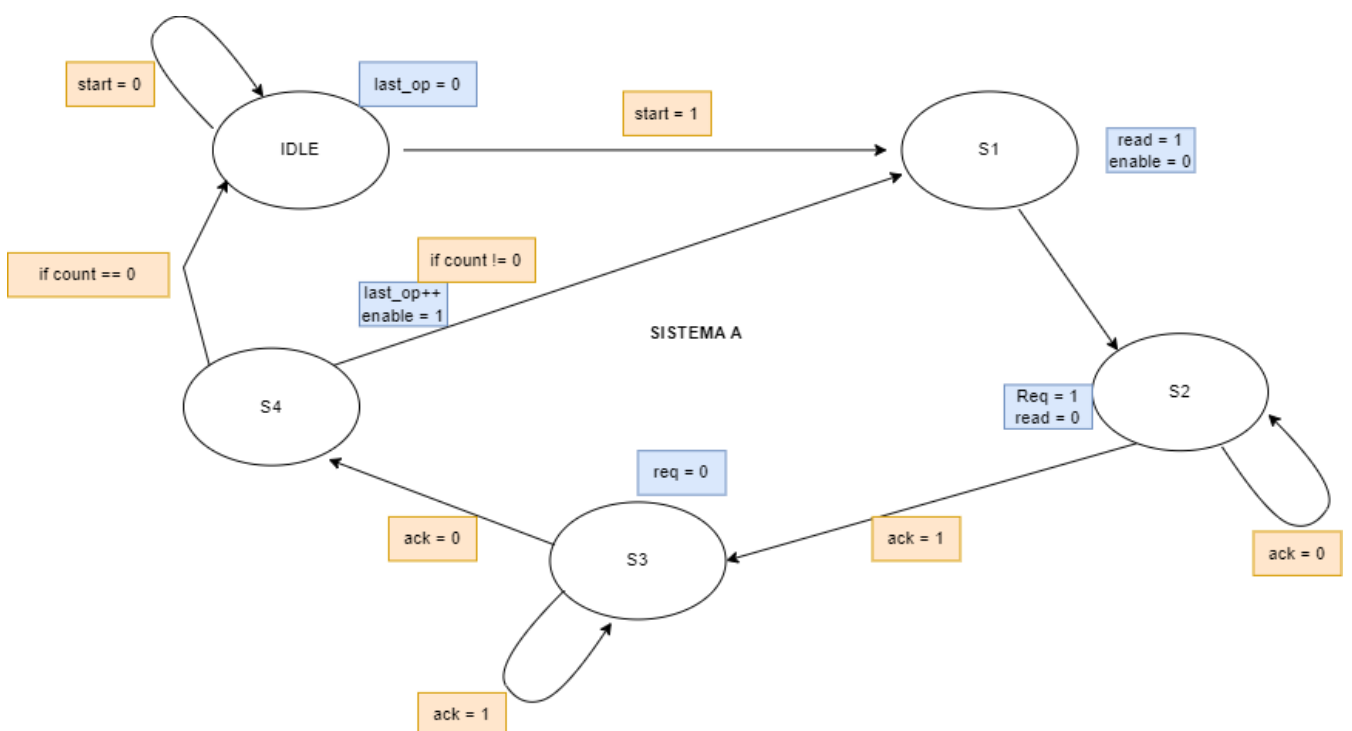
- Un'unità di controllo che comunica con l'unità di controllo del sistema A attraverso **req** e **ack** e che fornisce un segnale **enable** al proprio contatore, un segnale **load** al registro **op\_x** e un segnale **write** alla memoria.
- Un contatore mod-16 che riceve un segnale di abilitazione **enable** dall'unità di controllo e restituisce alla memoria il valore del conteggio, rappresentante l'indirizzo in cui leggere o scrivere.
- Una memoria che restituisce i valori scritti nelle sue locazioni come operando y al sommatore e riscrive nelle stesse posizioni il risultato fornito dal sommatore. Il clock della memoria è generato una OR tra i segnali **enable** e **load**, poiché deve attivarsi sia per le operazioni di scrittura che di lettura.
- Il registro **op\_x** riceve il segnale **load** dall'unità di controllo e il dato dal sistema A che pone successivamente in ingresso sommatore come operando x.

- Il sommatore riceve due operandi: uno proveniente da op\_x e l'altro dalla memoria. Successivamente, il sommatore calcola, appunto, la somma di questi due operandi e mette il risultato in uscita. Per questo modulo è stata utilizzata una semplice architettura dataflow nella quale si descrive il comportamento di un circuito digitale in termini del flusso dei dati.

Nel disegno successivo, viene illustrata l'architettura complessiva dei due sistemi.



Di seguito è possibile visualizzare il diagramma degli stati implementato dall'unità di controllo del sistema A.

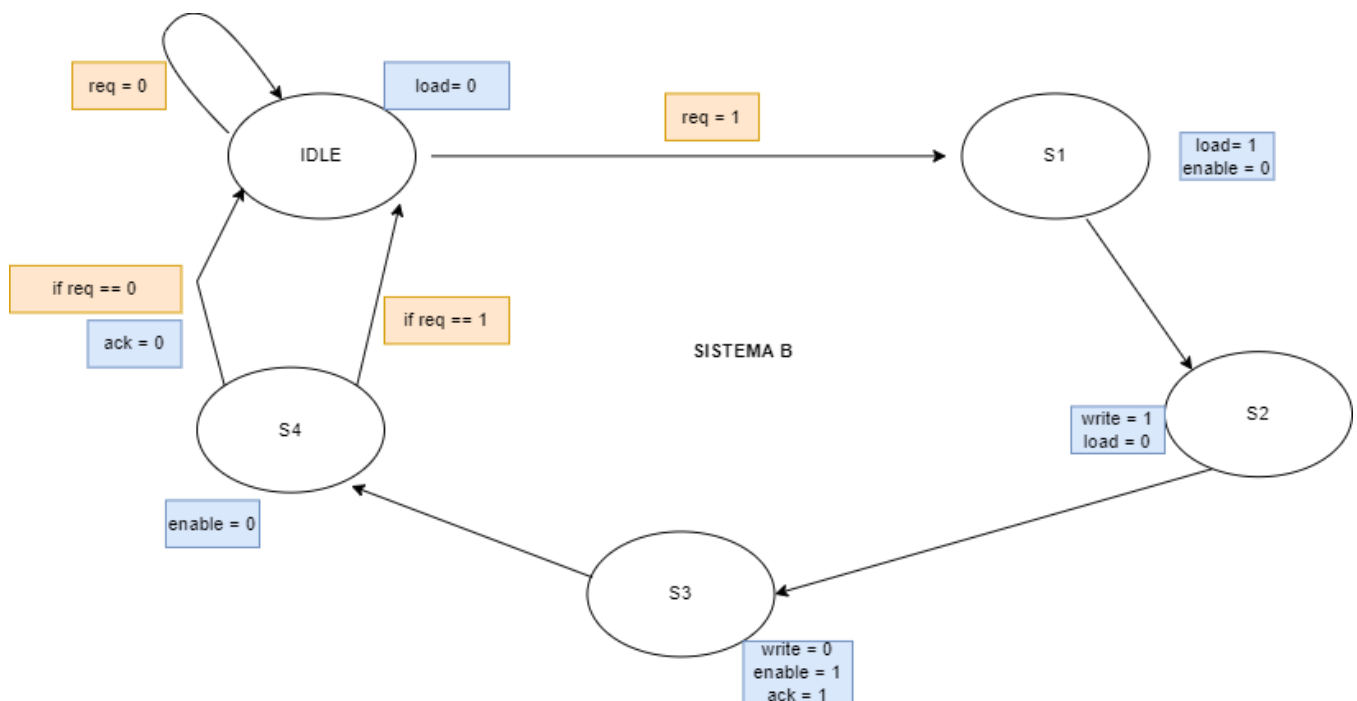


Si parte da uno stato **IDLE**. Il sistema non transita nello stato **S1** finché non riceve dall'esterno un segnale di **start**. Nello stato **S1**, viene abilitata la lettura (read) dalla ROM, che mette in uscita il dato conservato nella posizione data da address, mentre il segnale enable per il contatore è mantenuto a basso.

Nello stato **S2**, il segnale di read viene nuovamente abbassato e il segnale req viene alzato per avvisare il sistema B che il dato è stato inviato. Il sistema rimane in questo stato finché non osserva  $ack = 1$  che sta a indicare che B ha ricevuto i dati e li sta elaborando.

Nello stato **S3**, viene abbassato il segnale req. Si passa allo stato **S4** se anche  $ack = 0$ ; in questo stato, è possibile incrementare il contatore e ritornare allo stato **S1** per continuare la trasmissione, oppure, se il contatore si è di nuovo azzerato, si torna allo stato di **IDLE**.

L'immagine successiva rappresenta il diagramma degli stati finiti implementato dell'unità di controllo del sistema B.



Partendo dallo stato **IDLE**, il sistema mette load a 0 e rimane in questo stato fino a quando non vede il segnale req alto, che avvisa il sistema B del fatto che i dati ora scritti sui bus condivisi sono significativi.

Nello stato **S1**, il segnale load viene impostato a '1' (l'op\_x e la memoria caricano gli operandi in ingresso al sommatore) e enable è nuovamente azzerato.

Il sistema passa successivamente allo stato **S2**, dove write è alzato (il sommatore scrive il risultato della somma nella stessa posizione da cui si era ottenuto il secondo operando, sovrascrivendolo).

Nello stato **S3**, write viene abbassato e vengono alzati i segnali di enable (si incrementa il contatore) e di ack (si notifica A che B ha ricevuto correttamente i dati).

Nello stato **S4**, se  $req = 0$  il segnale ack viene abbassato (il sistema B è di nuovo disponibile per nuove richieste) e si va in **IDLE**, se, invece,  $req = 1$  il sistema torna direttamente in **IDLE** senza aggiornare il segnale ack.



## sistema.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sistema is -- complessivo
    Port (
        start, reset, clock : in std_logic ); -- clock e' il segnale di
tempificazione del sistema A (a 10 ns)
    end sistema;

architecture Structural of sistema is

    component sistema_B is
        Port (
            clock : in std_logic;
            reset : in std_logic;
            req : in std_logic;
            ack : out std_logic;
            data : in std_logic_vector(3 downto 0));
    end component;

    component sistema_A is
        Port (
            clock : in std_logic;
            start : in std_logic;
            reset : in std_logic;
            req : out std_logic;
            ack : in std_logic;
            data : out std_logic_vector(3 downto 0));
    end component;

    signal ack, req : std_logic;
    signal data : std_logic_vector(3 downto 0);

    signal count : integer range 0 to 1 := 0;
    signal clk_B : STD_LOGIC := '0'; -- clock e' il segnale di tempificazione
del sistema B (da 16 ns)
    constant CLK_DIVISION_FACTOR : integer := 2;
    constant COUNT_MAX : integer := CLK_DIVISION_FACTOR - 1;

begin

    sis_A : sistema_A port map (clock => clock, start => start, reset =>
reset, req => req, ack => ack, data => data);
```

```

    sis_B : sistema_B port map (clock => clk_B, reset => reset, req => req,
ack => ack, data => data);

process(clock, reset)
begin
    if reset = '1' then
        count <= 0;
        clk_B <= '0';
    elsif rising_edge(clock) then
        if count = COUNT_MAX then
            count <= 0;
            clk_B <= not clk_B;
        else
            count <= count + 1;
        end if;
    end if;
end process;

end architecture Structural;

```

---

## sistema\_A.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity sistema_A is
    Port (
        clock : in std_logic;
        start : in std_logic;
        reset : in std_logic;
        req : out std_logic;
        ack : in std_logic;
        data : out std_logic_vector(3 downto 0));
end sistema_A;

architecture Behavioral of sistema_A is
    component ROM is port (
        clock : in STD_LOGIC;
        reset : in STD_LOGIC;
        read : in STD_LOGIC;
        address : in STD_LOGIC_VECTOR(3 downto 0);
        data : out STD_LOGIC_VECTOR(3 downto 0));
    end component;

    component cont_mod_16 is
        generic (
            MAX_VALUE : integer := 15
        );

```

```

port (
    clock : in std_logic;
    reset : in std_logic;
    count : out integer range 0 to MAX_VALUE
);
end component;

component unita_controllo_A is
Port ( clock : in STD_LOGIC;
    start : in STD_LOGIC;
    read : out STD_LOGIC;
    reset : in STD_LOGIC;
    enable : out STD_LOGIC;
    count : in STD_LOGIC_VECTOR(3 downto 0);
    req : out std_logic;
    ack : in std_logic);
end component;

signal count : std_logic_vector(3 downto 0) := (others=>'0');
signal enable : std_logic := '0';
signal read : std_logic := '0';
signal conteggio : integer := 0;

begin
    mem_rom : ROM port map (clock => clock, reset => reset, read => read,
address => count, data => data);

    cont : cont_mod_16 port map(clock => enable, reset => reset, count =>
conteggio);

    count <= std_logic_vector(to_unsigned(conteggio, count'length));

    control_unit : unita_controllo_A port map (clock => clock, read => read,
start => start, reset => reset, enable => enable, count => count, req => req,
ack => ack);

end Behavioral;

```

---

## unita\_controllo\_A.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity unita_controllo_A is
    Port ( clock : in STD_LOGIC;
        start : in STD_LOGIC;
        reset : in STD_LOGIC;
        read : out STD_LOGIC;

```

```

        enable : out STD_LOGIC;
        count : in STD_LOGIC_VECTOR(3 downto 0);
        req : out std_logic;
        ack : in std_logic);
end unita_controllo_A;

architecture Behavioral of unita_controllo_A is

    type state is (idle, s1, s2, s3, s4);
    signal current_state, next_state: state;

begin
    reg_stato: process(clock)
    begin
        if(clock'event and clock='1') then
            if(reset='1') then
                current_state <= idle;
            else
                current_state <= next_state;
            end if;
        end if;
    end process;

    comb: process(current_state, start, count, ack)
    variable last_op : integer;
    begin
        case current_state is

            when idle =>
                last_op := 0;
                if(start='1') then
                    next_state <= s1;
                else
                    next_state <= idle;
                end if;

            when s1 =>
                read <= '1';
                enable <= '0';
                next_state <= s2;

            when s2 =>
                req <= '1';
                read <= '0';
                if(ack/='1') then
                    next_state <= s2;
                else next_state <= s3;
                end if;

            when s3 =>

```

```

        req <= '0';
        if(ack='1') then
            next_state <= s3;
        else
            next_state <= s4;
        end if;

    when s4 =>
        if(count="0000" and last_op/=0) then
            next_state <= idle;
        else
            last_op:=1;
            enable <= '1';
            next_state <= s1;
        end if;

    end case;
end process;

end Behavioral;

```

---

## ROM. vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity ROM is port (
    clock : in STD_LOGIC;
    read : in STD_LOGIC;
    reset : in STD_LOGIC;
    address : in STD_LOGIC_VECTOR(3 downto 0);
    data : out STD_LOGIC_VECTOR(3 downto 0));
end ROM;

architecture Behavioral of ROM is
    type memory_array is array (0 to 15) of STD_LOGIC_VECTOR(3 downto 0);
    constant rom_data : memory_array := (
        0 => "0000", 1 => "0001", 2 => "0010", 3 => "0011",
        4 => "0100", 5 => "0101", 6 => "0110", 7 => "0111",
        8 => "1000", 9 => "1001", 10 => "1010", 11 => "1011",
        12 => "1100", 13 => "1101", 14 => "1110", 15 => "1111"
    );
begin
    process(clock)
    begin
        if reset='1' then
            data <= (others=>'0');
        elsif(rising_edge(clock) and read='1') then

```

```

        data <= rom_data(to_integer(unsigned(address)));
    end if;
end process;
end Behavioral;

```

---

## cont\_mod\_16.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity cont_mod_16 is
    generic (
        MAX_VALUE : integer := 15
    );
    port (
        clock : in std_logic;
        reset : in std_logic;
        count : out integer range 0 to MAX_VALUE
    );
end entity;

architecture Behavioral of cont_mod_16 is

    signal counter : integer range 0 to MAX_VALUE := 0;

begin

    process(clock, reset)
    begin
        if reset = '1' then
            counter <= 0;
        elsif rising_edge(clock) then
            if counter < MAX_VALUE then
                counter <= counter + 1;
            else
                counter <= 0;
            end if;
        end if;
    end process;

    count <= counter;

end architecture;

```

---

## sistema\_B.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

```

```

entity sistema_B is
  Port (
    clock : in std_logic;
    reset : in std_logic;
    req : in std_logic;
    ack : out std_logic;
    data : in std_logic_vector(3 downto 0));
end sistema_B;

architecture Structural of sistema_B is

  component cont_mod_16 is
    generic (
      MAX_VALUE : integer := 15
    );
    port (
      clock : in std_logic;
      reset : in std_logic;
      count : out integer range 0 to MAX_VALUE
    );
  end component;

  component operando_x is
    Port (
      clock : in std_logic;
      reset : in std_logic;
      dato_in : in std_logic_vector(3 downto 0);
      dato_out : out std_logic_vector(3 downto 0)
    );
  end component;

  component sommatore is
    Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
           B : in STD_LOGIC_VECTOR (3 downto 0);
           sum : out STD_LOGIC_VECTOR (3 downto 0));
  end component;

  component unita_controllo_B is
    Port ( clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           load : out STD_LOGIC;
           write : out STD_LOGIC;
           enable : out STD_LOGIC;
           req : in std_logic;
           ack : out std_logic);
  end component;

  component memoria is
    Port (

```

```

        reset : in STD_LOGIC;
        clock : in STD_LOGIC;
        address : in STD_LOGIC_VECTOR (3 downto 0);
        data_in : in STD_LOGIC_VECTOR (3 downto 0);
        write : in STD_LOGIC;
        load : in STD_LOGIC;
        data_out : out STD_LOGIC_VECTOR (3 downto 0)
    );
end component;

signal load, write : std_logic := '0';
signal count : std_logic_vector(3 downto 0) := (others=>'0');
signal enable : std_logic := '0';
signal conteggio : integer := 0;
signal op_x, op_y, sum: std_logic_vector(3 downto 0) := (others=>'0');
signal clock_mem : std_logic := '0';

begin
    control_unit_B : unita_controllo_B port map (clock => clock, enable =>
enable, reset => reset, load => load, write => write, req => req, ack =>
ack);

    cont_B : cont_mod_16 port map(clock => enable, reset => reset, count =>
conteggio);

    count <= std_logic_vector(to_unsigned(conteggio, count'length));

    operand_x : operando_x port map(clock => load, reset => reset, dato_in =>
data, dato_out => op_x);

    somm : sommatore port map (A => op_x, B => op_y, sum => sum);

    clock_mem <= enable or load;

    mem : memoria port map(reset => reset, clock => clock_mem, address =>
count, data_in => sum, write => write, load => load, data_out => op_y);
end Structural;

```

---

## unita\_controllo\_B.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity unita_controllo_B is
    Port ( clock : in STD_LOGIC;
        reset : in STD_LOGIC;
        enable : out STD_LOGIC;
        load : out STD_LOGIC;

```



```

        write : out STD_LOGIC;
        req : in std_logic;
        ack : out std_logic);
end unita_controllo_B;

architecture Behavioral of unita_controllo_B is

    type state is (idle, s1, s2, s3, s4);
    signal current_state: state;

begin

    process(clock)
    begin
        if rising_edge(clock) then
            if reset = '1' then
                current_state <= idle;
            else
                case current_state is
                    when idle =>
                        load <= '0';
                        if req = '1' then
                            current_state <= s1;
                        else
                            current_state <= idle;
                        end if;

                    when s1 =>
                        load <= '1';
                        enable <= '0';
                        current_state <= s2;

                    when s2 =>
                        load <= '0';
                        write <= '1';
                        current_state <= s3;

                    when s3 =>
                        write <= '0';
                        enable <= '1';
                        ack <= '1';
                        current_state <= s4;

                    when s4 =>
                        enable <= '0';
                        if req = '1' then
                            current_state <= idle;
                        else
                            ack <= '0';
                            current_state <= idle;
                        end if;
                    end case;
                end if;
            end if;
        end if;
    end process;
end architecture Behavioral of unita_controllo_B;

```

```

        end if;
    end case;
end if;
end if;
end process;

end Behavioral;

```

---

## operando\_x.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity operando_x is
    Port (
        clock : in std_logic;
        reset : in std_logic;
        dato_in : in std_logic_vector(3 downto 0);
        dato_out : out std_logic_vector(3 downto 0)
    );
end operando_x;

architecture Behavioral of operando_x is

    signal op_x : std_logic_vector(3 downto 0) := (others => '0');

begin
    process(clock, reset)
    begin
        if reset = '1' then
            op_x <= (others => '0');
        elsif (clock'event and clock='1') then
            op_x <= dato_in;
        end if;
    end process;

    dato_out <= op_x;
end Behavioral;

```

---

## sommatore.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity sommatore is
    Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
          B : in STD_LOGIC_VECTOR (3 downto 0);
          sum : out STD_LOGIC_VECTOR (3 downto 0));

```

```

end sommatore;

architecture Dataflow of sommatore is
begin
    sum <= std_logic_vector(unsigned(A) + unsigned(B));
end Dataflow;

```

---

## memoria.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity memoria is
    Port (
        reset : in STD_LOGIC;
        clock : in STD_LOGIC;
        address : in STD_LOGIC_VECTOR (3 downto 0);
        data_in : in STD_LOGIC_VECTOR (3 downto 0);
        write : in STD_LOGIC;
        load : in STD_LOGIC;
        data_out : out STD_LOGIC_VECTOR (3 downto 0)
    );
end memoria;

architecture Behavioral of memoria is
    type memory_array is array (0 to 15) of STD_LOGIC_VECTOR (3 downto 0);
    signal memory : memory_array := (
        0 => "0000", 1 => "0001", 2 => "0010", 3 => "0011",
        4 => "0100", 5 => "0101", 6 => "0110", 7 => "0111",
        8 => "1000", 9 => "1001", 10 => "1010", 11 => "1011",
        12 => "1100", 13 => "1101", 14 => "1110", 15 => "1111"
    );
begin
    process(clock, reset)
    begin
        if reset='1' then
            data_out <= (others=>'0');
        elsif (clock'event and clock='1') then
            if write = '1' then
                memory(to_integer(unsigned(address))) <= data_in;
            elsif load = '1' then
                data_out <= memory(to_integer(unsigned(address)));
            end if;
        end if;
    end process;
end Behavioral;

```

---

Sono stati sviluppati tre testbench principali: uno dedicato alla simulazione del sistema A in isolamento (testbench\_A), uno per testare esclusivamente B, e infine, uno per valutare l'intero sistema composto sia da A che da B (testbench\_sis). In seguito, viene riportato sia il codice VHDL che i risultati ottenuti durante l'ultima simulazione menzionata.

---

#### **testbench\_sis.vhd**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity testbench_sis is
end testbench_sis;

architecture Behavioral of testbench_sis is
    component sistema is
        Port (
            start, reset, clock : in std_logic );
        end component;

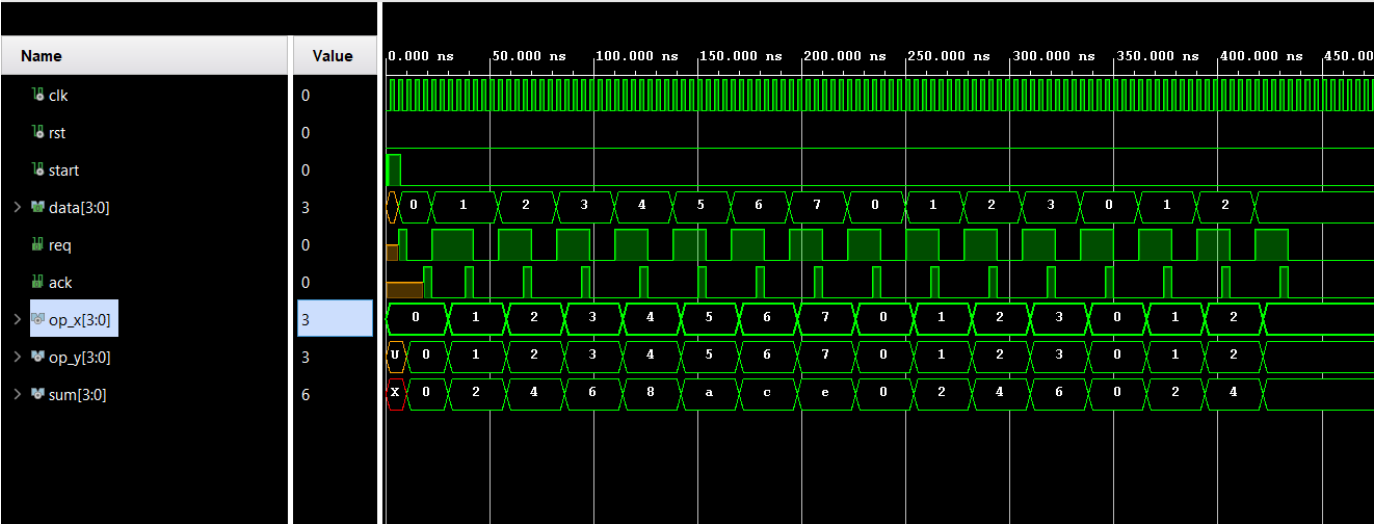
    signal clk, rst, start : std_logic := '0';
begin

    clock_gen: process
    begin
        wait for 2 ns;
        clk <= not clk;
    end process;

    sis : sistema port map (start=>start, reset=>rst, clock=>clk);

    process
    begin
        wait for 1 ns;
        start <= '1';
        wait for 6 ns;
        start <= '0';
        wait for 900 ns;
        wait;
    end process;

end Behavioral;
```



## Capitolo 5: Processore

### Esercizio 9: Processore

La **MIC-1**, creata da Andrew S. **Tanenbaum**, è macchina virtuale che implementa la **JVM** ovvero la Java Virtual Machine che opera esclusivamente su numeri interi. Il MIC-1 rappresenta un classico esempio di un sistema digitale basata su controllo microprogrammato anziché cablato.

L'architettura del MIC-1 è composta da vari componenti principali, in primis il **datapath**. Il percorso dati è, in generale, il sistema di circuiti che esegue le operazioni richieste dalle istruzioni del processore; nel caso del MIC-1 esso è formato dagli seguenti elementi:

- **Registri**

- **MAR** (Memory Address Register): Questo registro contiene la posizione in memoria a cui accedere per svolgere una lettura o una scrittura di dati. Quando viene inserito un indirizzo nel MAR, i dati corrispondenti a quell'indirizzo vengono caricati nella MDR.
- **MDR** (Memory Data Register): Questo registro contiene i dati che si leggono o scrivono in memoria nella posizione data dal MAR.
- **PC** (Program Counter): Questo registro contiene l'indirizzo della prossima istruzione da eseguire. Dopo l'esecuzione di un'istruzione, il PC viene incrementato di uno per puntare alla istruzione successiva.
- **MBR** (Memory Buffer Register): Questo registro contiene la porzione da esaminare dell'istruzione corrente; questo registro è l'unico ad essere da 8 bit mentre tutti gli altri sono da 32 bit.
- **SP** (Stack Pointer): Questo registro contiene un puntatore all'elemento in cima allo stack.
- **LV** (Local Variable): Questo registro contiene un riferimento alla base sullo stack della **Local Variable Frame** del metodo attualmente in esecuzione. La Local Variable Frame contiene le variabili locali del metodo ed è sovrastata da l'**Operand Stack** il cui ultimo operando è proprio quello puntato da SP.
- **CPP** (Constant Pool Pointer): Questo registro contiene l'indirizzo della base dall'area di memoria nota come **Constant Pool** la quale contiene le costanti, le stringhe e i puntatori del programma attivo.
- **TOS** (Top Of Stack): Questo registro contiene una copia l'elemento in cima allo stack. Quando un nuovo valore diventa quello superiore sullo stack, il TOS viene aggiornato.
- **OPC**: Questo registro viene utilizzato per contenere risultati intermedi delle operazioni.
- **H** (Hold): Quando si esegue un'operazione aritmetica su due operandi, uno dei due viene prima trasferito nel registro H; l'altro invece viene preso direttamente dal bus al momento del calcolo.

- **Bus:**

- **B**: Questo bus è utilizzato per trasferire i dati tra i registri e l'ALU. Tutti i registri menzionati in precedenza, ad eccezione di MAR, sono dotati di un segnale di controllo che gestisce la scrittura sul bus B. Si noti che in un dato istante, solo un registro può scrivere sul bus B.
- **C**: Questo bus è impiegato per trasferire i dati tra lo shift register che contiene il risultato dell'ALU e i registri. Tutti i registri, tranne MBR, sono dotati di un segnale di controllo che permette la lettura dal bus C.

- **ALU** (Arithmetic Logic Unit): Questo modulo esegue le operazioni aritmetiche e logiche. Nella MIC-1, l'ALU ha sei linee di controllo, le quali determinano quale funzione eseguire e quali input utilizzare.
  - **F0** e **F1** selezionano la funzione dell'ALU: AND, OR, NOT o ADD.

- **INVA** determina se eseguire o meno il complemento a uno, ossia la negazione, dell'operando proveniente dal registro H.
- **ENA** attiva l'acquisizione del primo operando da H, altrimenti esso è zero.
- **ENB** attiva l'acquisizione del secondo operando dal bus B, altrimenti esso è zero.
- **INC** specifica se incrementare di uno il risultato ottenuto o mantenerlo invariato.

L'ALU ha anche due bit di output di stato: **N** e **Z** che indicando rispettivamente se il risultato dell'operazione è negativo oppure zero. I valori di questi flag sono memorizzati all'interno di due flip-flop, rendendoli facilmente accessibili.

Il risultato viene immagazzinato in uno shift register prima di essere trasferito sul bus C. Lo **shifter** è dotato di due segnali di controllo

- **SLL8** (Shift Left Logical): Quando questa linea è attiva, lo shifter esegue un'operazione di shift a sinistra logico, spostando tutti i bit verso sinistra di 8 posizioni e riempiendo i bit più significativi con degli zeri.
- **SRA1** (Shift Right Arithmetic): Quando questa linea è alta, lo shifter esegue un'operazione di shift a destra aritmetico ovvero sposta tutti i bit verso destra di 1 posizione e mantenendo il bit più significativo uguale al suo valore originale, zero o uno.

La MIC-1 dispone di due modalità di comunicazione con la memoria:

1. Un terminale a 32 bit controllato dai registri MAR e MDR: il MAR indirizza le word di dati, mentre il MDR gestisce la memorizzazione o l'acquisizione di tali dati.
2. Un terminale ad 8 bit controllato dai registri PC e MBR: il PC indirizza singoli byte di programma, e l'MBR è coinvolto nella lettura e scrittura di tali byte. Inoltre, oltre al segnale che gestisce la scrittura sul bus B, l'MBR riceve in ingresso un ulteriore segnale di controllo che determina se il byte deve essere trasmesso su B come un vettore di bit con segno o senza.

L'operazione di lettura o scrittura della memoria, effettuata con uno dei metodi sopra menzionati, si svolge in due fasi distinte. Inizialmente, si inserisce un indirizzo nei registri PC o MAR, avviando così la lettura o la scrittura. Tuttavia, tale operazione viene completata solo durante il ciclo di clock successivo, quando avviene l'effettivo caricamento dei dati da o verso il MBR o MDR.

Nel complesso, considerando che alcuni di essi sono mutualmente esclusivi, il datapath è guidato nel suo flusso di esecuzione da 24 segnali di controllo che vengono generati dal processore.

Nell'architettura della IJVM di Tanenbaum ad ogni istruzione ISA è associata una sequenza di microistruzioni da eseguire detta micro-routine. Tutte le possibili microistruzioni, chiamate anche **control word**, hanno una lunghezza fissa di 36 bit e sono memorizzate all'interno di una memoria ovvero la **Control Store** che può contenere fino a 512 di queste parole.

L'accesso alla Control Store avviene attraverso due registri chiave: il **MPC** (Micro Program Counter) e il **MIR** (Micro Instruction Register). Il MPC è un registro virtuale da 9 bit che stabilisce l'indirizzo da cui leggere la prossima microistruzione, mentre il MIR contiene la control word letta dalla posizione indicata in precedenza dal MPC.

Per semplicità, la posizione della prima microistruzione di un'istruzione è data proprio dal valore in binario dell'**opcode** corrispondente a quell'istruzione. Successivamente, nonostante le control word siano memorizzate "in ordine" nella Control Store, la loro esecuzione può seguire un percorso diverso da quello prestabilito in base a diversi fattori.

I primi 8 bit della microistruzione (**Addr**) nel registro MIR rappresentano l'ipotetico indirizzo della successiva microistruzione, mentre i successivi 3 bit (**JMPC**, **JAMN**, **JAMZ**) contengono codici di condizione per i salti. È

in base a questi bit e ad altre informazioni provenienti dal datapath che si decide come aggiornare il MPC. Precisamente, vengono impiegati due blocchi logici:

1. **High bit:** Questo circuito stabilisce il bit più significativo di MPC in base al bit più significativo di Addr, ai bit JAMN, JAMZ e ai flag N e Z precedentemente presentati, secondo la seguente funzione.

$$\text{MPC}[0] = \text{Addr}[0] + Z \cdot \text{JAMZ} + N \cdot \text{JAMN}$$

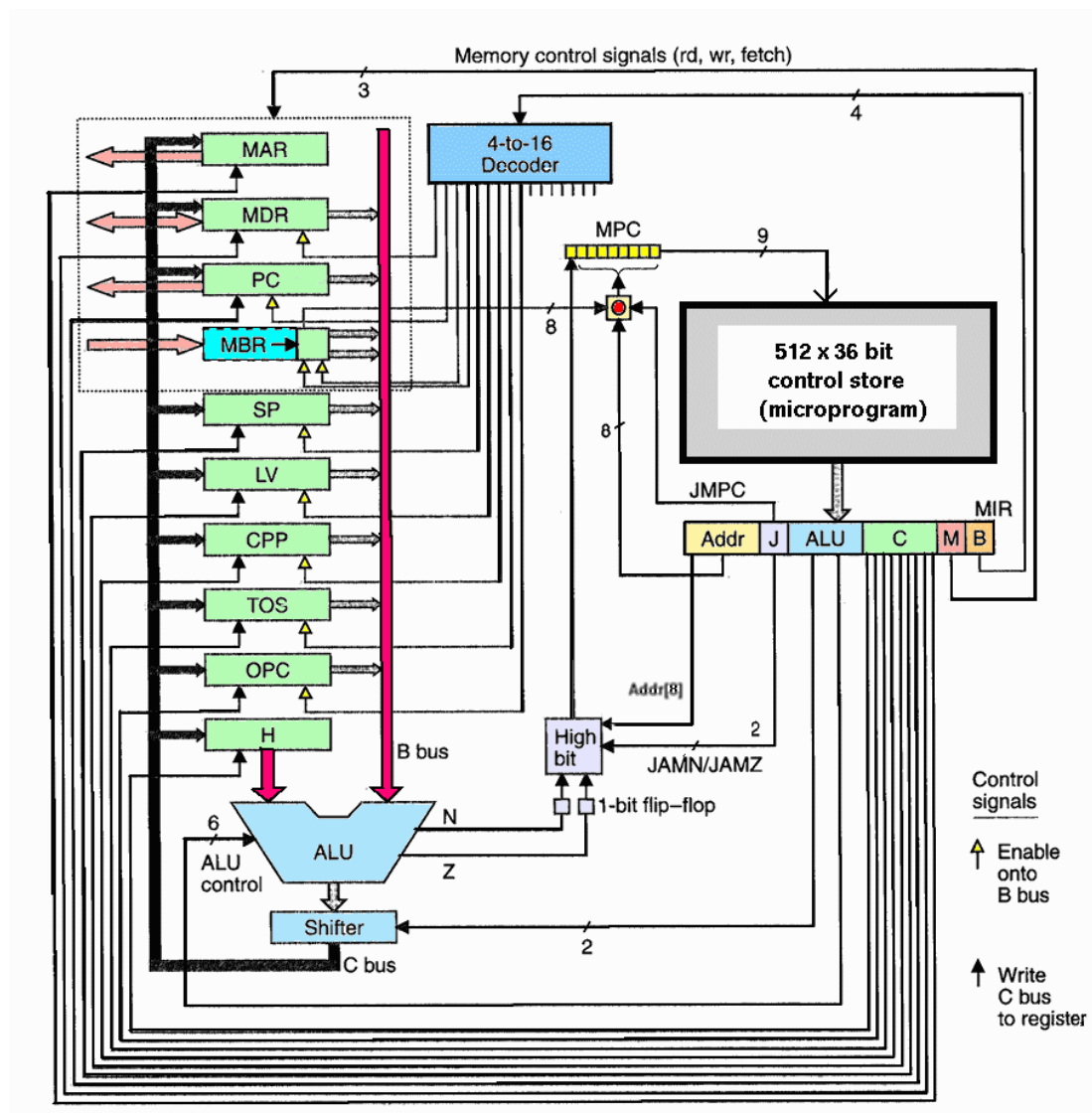
2. **Jump logic:** Questo blocco calcola gli 8 bit meno significativi del nuovo contenuto di MPC utilizzando gli 8 bit meno significativi di Addr, l'intero contenuto del registro MBR e il codice JMPCL. L'espressione da calcolare è riportata successivamente.

$$MPC[1:8] = \begin{cases} Addr[1:8] & se\ JMPC = 0 \\ Addr[1:8] + MBR[0:7] & se\ JMPC = 1 \end{cases}$$

Se JMPC è alto, ci si aspetta che tutti i bit di Addr[1:8] siano zero; in questo caso quindi i bit MPC[1:8] sono posti proprio a MBR[0:7], che tipicamente rappresenta un codice operativo.

Invece, i restanti bit della control word, ovvero quelli successivi a JAMZ, rappresentano i segnali di controllo del percorso dei dati. Si osservi che i segnali che gestiscono la scrittura dei registri sul bus B sono codificati su 4 bit, poiché non sono necessari 9 segnali distinti, essendo questi mutualmente esclusivi.

Nell'immagine seguente è illustrata l'architettura complessiva del MIC-1.





Per definire la sequenza di microistruzioni che implementa un'istruzione ISA, è possibile utilizzare il linguaggio **MAL** (Micro Assembly Language). Si noti che nel microcodice scritto in questo linguaggio simbolico il termine "fetch" indica la comunicazione con la memoria tramite i registri PC e MBR, mentre il termine "read" si riferisce alla comunicazione che coinvolge MAR e MDR.

L'istruzione **isub** è un'istruzione ISA che estrai due numeri interi del top dello stack, esegue la sottrazione tra di essi e fa il caricamento del risultato sulla cima dello stack. Chiaramente prima di eseguire isub, è necessario aver caricato almeno due operandi sullo stack, ad esempio, tramite due istruzioni di bipush. Il **bipush** esegue il push di un byte fornito come operando all'istruzione sullo stack.

Il microcodice MAL associato all'istruzione **isub** è il seguente.

---

```
isub = 0x5C:
    MAR = SP = SP - 1; rd
    H = TOS
    MDR = TOS = MDR - H; wr; goto main
```

---

"0x5C" è il codice operativo dell'istruzione ISA.

1. **MAR = SP = SP - 1; rd**: Viene decrementato di 1 il registro e il valore risultante viene scritto anche in MAR. SP non punta più quindi alla cima dello stack ma all'elemento precedente che al momento contiene il primo operando della sottrazione e che alla fine conterrà il risultato. "**rd**" indica che è stata fatta partire un'operazione di lettura della memoria in corrispondenza della posizione data dal MAR, ovvero si sta leggendo il primo operando.
2. **H = TOS**: Viene copiato nel registro H il valore attualmente presente al top dello stack, corrispondente al secondo operando, poiché il registro TOS non è stato ancora aggiornato.
3. **MDR = TOS = MDR - H; wr; goto main**: Effettua la sottrazione tra i registri MDR (il primo operando poiché a questo punto si è conclusa la lettura) e H (il secondo operando) e il risultato viene salvato sia in MDR che in TOS. "**wr**" indica che è in corso un'operazione di scrittura del valore di MDR, ovvero il risultato della differenza, nell'indirizzo ancora mantenuto in MAR che chiaramente non è stato necessario modificare. Infine, l'istruzione "**goto**" consente di eseguire un salto incondizionato all'etichetta "main".

Per simulare l'istruzione isub è stato modificato il file "program.ajvm" come segue.

---

```
.main
    .var
    a
    .endvar

    BIPUSH 0xA
    BIPUSH 0x5
    ISUB
    ISTORE a
    HALT
    .endmethod
```

---

In questo programma, vengono caricati i valori 0xA e 0x5 sulla pila tramite le istruzioni BIPUSH. Successivamente, l'istruzione ISUB sottrae il secondo valore dal primo, modificando così lo stack da [0xA, 0x5] a [0x5]. L'istruzione **ISTORE** memorizza quindi il risultato, 0x5, nella variabile "a". Infine, l'istruzione **HALT** termina l'esecuzione del programma.

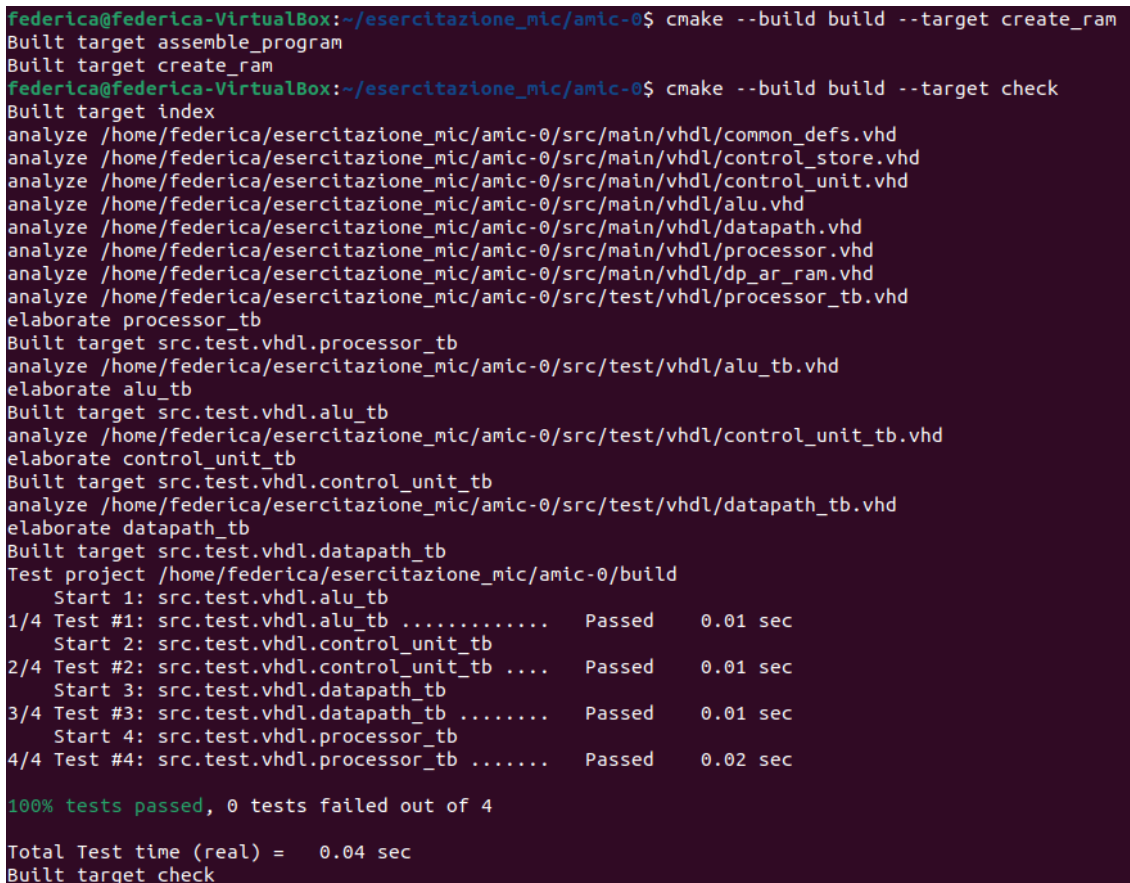
Tra i testbench forniti, l'unico che richiede modifiche è "processor\_tb.vhd"; è infatti necessario adattare il processo "wavegen\_proc".

---

```
-- Waveform generation
wavegen_proc: process
begin
    wait until clk = '1';
    wait for 2 ns;
    reset <= '1';
    wait for 10 ns;
    reset <= '0';
    wait until mem_instr_addr = x"0000000A" and mem_data_we = '1';
    assert mem_data_out = x"00000005" report "Bad calculated value" severity failure;
    wait until mem_instr_addr = x"0000000B";
end_run := true;
    wait;
end process wavegen_proc;
```

---

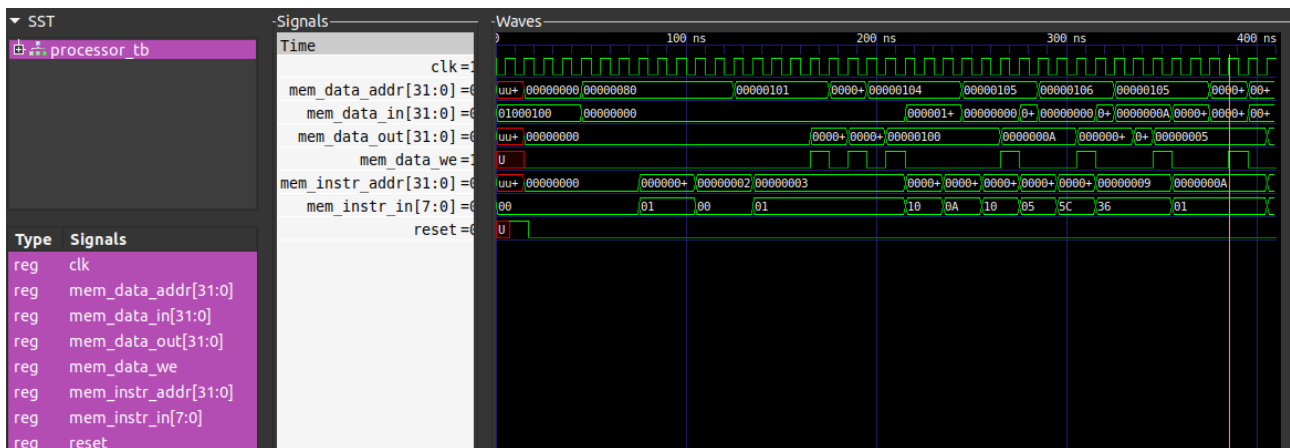
Le immagini sottostanti mostrano i risultati positivi dei test



```
federica@federica-VirtualBox:~/esercitazione_mic/amic-0$ cmake --build build --target create_ram
Built target assemble_program
Built target create_ram
federica@federica-VirtualBox:~/esercitazione_mic/amic-0$ cmake --build build --target check
Built target index
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/common_defs.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/control_store.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/control_unit.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/alu.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/datapath.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/processor.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/dp_ar_ram.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/test/vhdl/processor_tb.vhd
elaborate processor_tb
Built target src.test.vhdl.processor_tb
analyze /home/federica/esercitazione_mic/amic-0/src/test/vhdl/alu_tb.vhd
elaborate alu_tb
Built target src.test.vhdl.alu_tb
analyze /home/federica/esercitazione_mic/amic-0/src/test/vhdl/control_unit_tb.vhd
elaborate control_unit_tb
Built target src.test.vhdl.control_unit_tb
analyze /home/federica/esercitazione_mic/amic-0/src/test/vhdl/datapath_tb.vhd
elaborate datapath_tb
Built target src.test.vhdl.datapath_tb
Test project /home/federica/esercitazione_mic/amic-0/build
  Start 1: src.test.vhdl.alu_tb
1/4 Test #1: src.test.vhdl.alu_tb ..... Passed    0.01 sec
  Start 2: src.test.vhdl.control_unit_tb
2/4 Test #2: src.test.vhdl.control_unit_tb .... Passed    0.01 sec
  Start 3: src.test.vhdl.datapath_tb
3/4 Test #3: src.test.vhdl.datapath_tb ..... Passed    0.01 sec
  Start 4: src.test.vhdl.processor_tb
4/4 Test #4: src.test.vhdl.processor_tb ..... Passed    0.02 sec

100% tests passed, 0 tests failed out of 4

Total Test time (real) =  0.04 sec
Built target check
```



L'istruzione **swap** effettua la rimozione di due numeri dalla cima dello stack e successivamente li reinserisce sulla pila, ma in ordine inverso. Il microcodice MAL associato all'istruzione **swap** è scritto successivamente.

```

swap = 0x5F:
    MAR = SP - 1; rd
    MAR = SP
    H = MDR; wr
    MDR = TOS
    MAR = SP - 1; wr
    TOS = H; goto main

```

"0x5F" è l'opcode dell'istruzione ISA.

1. **MAR = SP - 1; rd**: Viene letto dalla memoria il valore nella posizione immediatamente sotto la cima dello stack. Solo successivamente il dato letto, ovvero il primo dei due elementi da scambiare, viene memorizzato nel registro MDR.
2. **MAR = SP**: Aggiorna il registro MAR in modo che punti al top dello stack, ovvero al secondo dei due numeri da scambiare.
3. **H = MDR; wr**: Viene scritto il valore contenuto in MDR, quindi il primo numero, nel registro temporaneo H. Si fa, poi, partire la lettura del secondo numero dato che ora, poiché il registro MAR è stato opportunamente configurato a puntare al TOS.
4. **MDR = TOS**: Carica l'elemento attualmente presente nel registro TOS, ovvero il secondo numero, in MDR.
5. **MAR = SP - 1; wr**: Aggiorna MAR a puntare nuovamente alla posizione sotto la cima dello stack. Si dà il via a un'operazione di scrittura dei dati in MDR, ossia il secondo valore, all'indirizzo dato da MAR quindi nella posizione sotto il top.
6. **TOS = H; goto main**: Si scrive il contenuto di H (il primo elemento) sul top dello stack completando così lo swap. Infine, c'è un salto all'etichetta main.

Per emulare l'istruzione swap, il file "program.ajvm" riscritto nel seguente modo.

```

.main
.var
a
.endvar
BIPUSH 0x1
BIPUSH 0x2
SWAP

```

```
ISTORE a
HALT
.endmethod
```

---

Prima dell'operazione di scambio la pila degli operandi è [0x1, 0x2], dopo diventa [0x2, 0x1]; quindi alla fine del programma la variabile "a" contiene 0x2.

Come in precedenza, si modifica il processo con label "wavegen\_proc" del file "processor\_tb.vhd" al fine di testare specificamente questo aspetto.

---

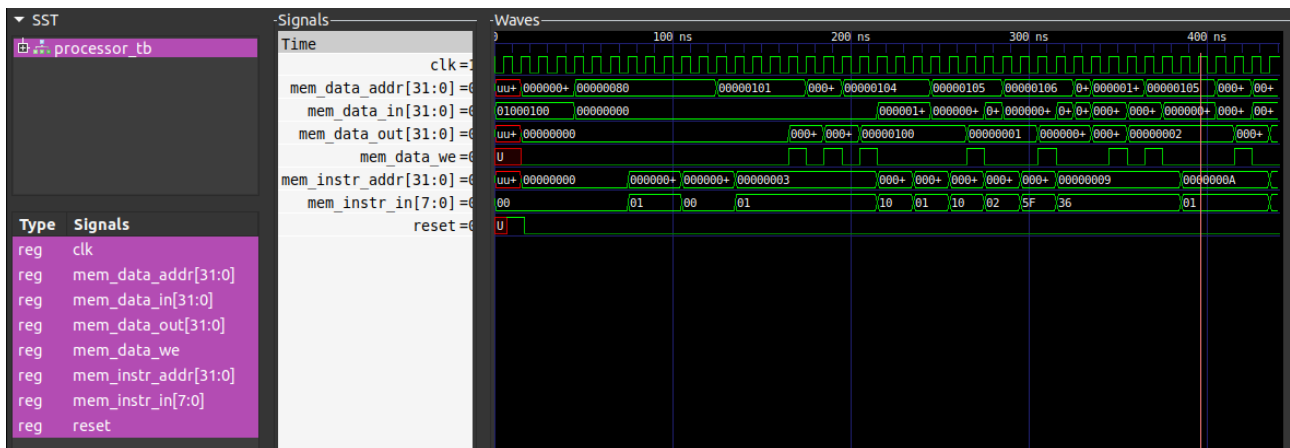
```
-- Waveform generation
wavegen_proc: process
begin
    wait until clk = '1';
    wait for 2 ns;
    reset <= '1';
    wait for 10 ns;
    reset <= '0';
    wait until mem_instr_addr = x"0000000A" and mem_data_we = '1';
    assert mem_data_out = x"00000001" report "Bad calculated value" severity failure;
    wait until mem_instr_addr = x"0000000B";
end_run := true;
    wait;
end process wavegen_proc;
```

---

```
federica@federica-VirtualBox:~/esercitazione_mic/amic-0$ cmake --build build --target create_ram
Built target assemble_program
Built target create_ram
federica@federica-VirtualBox:~/esercitazione_mic/amic-0$ cmake --build build --target check
Built target index
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/common_defs.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/control_store.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/control_unit.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/alu.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/datapath.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/processor.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/dp_ar_ram.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/test/vhdl/processor_tb.vhd
elaborate processor_tb
Built target src.test.vhdl.processor_tb
analyze /home/federica/esercitazione_mic/amic-0/src/test/vhdl/alu_tb.vhd
elaborate alu_tb
Built target src.test.vhdl.alu_tb
analyze /home/federica/esercitazione_mic/amic-0/src/test/vhdl/control_unit_tb.vhd
elaborate control_unit_tb
Built target src.test.vhdl.control_unit_tb
analyze /home/federica/esercitazione_mic/amic-0/src/test/vhdl/datapath_tb.vhd
elaborate datapath_tb
Built target src.test.vhdl.datapath_tb
Test project /home/federica/esercitazione_mic/amic-0/build
  Start 1: src.test.vhdl.alu_tb
1/4 Test #1: src.test.vhdl.alu_tb ..... Passed    0.01 sec
  Start 2: src.test.vhdl.control_unit_tb
2/4 Test #2: src.test.vhdl.control_unit_tb .... Passed    0.01 sec
  Start 3: src.test.vhdl.datapath_tb
3/4 Test #3: src.test.vhdl.datapath_tb ..... Passed    0.01 sec
  Start 4: src.test.vhdl.processor_tb
4/4 Test #4: src.test.vhdl.processor_tb ..... Passed    0.03 sec

100% tests passed, 0 tests failed out of 4

Total Test time (real) =  0.05 sec
Built target check
federica@federica-VirtualBox:~/esercitazione_mic/amic-0$
```



L'istruzione **iand** fa il pop di due elementi dello stack, esegue l'operazione logica di AND tra i due e fa il push del risultato sulla pila.

```
iand = 0x7E:
MAR = SP = SP - 1; rd
H = TOS
MDR = TOS = MDR AND H; wr; goto main
```

Adesso si modifica quest'istruzione, nel file "ajvm.mal", in modo che anziché effettuare una AND, venga eseguita un'operazione di OR.

```
iand = 0x7E:
MAR = SP = SP - 1; rd
H = TOS
MDR = TOS = MDR OR H; wr; goto main
```

1. **MAR = SP = SP - 1; rd:** L'SP viene modificato in modo tale da puntare all'elemento prima del top dello stack; l'indirizzo ricavato viene scritto anche nel registro MAR e si fa partire una lettura in questa posizione che contiene uno dei due operandi.
2. **H = TOS:** Il valore che al momento si trova sulla cima della pila, ovvero l'altro operando, viene memorizzato in H.
3. **MDR = TOS = MDR OR H; wr; goto main:** Al posto dell'operazione logica di AND, viene eseguita un'operazione di OR tra gli elementi presenti nei registri MDR (che ora contiene il primo operando) e H. Il risultato viene scritto sia in TOS che in MBR e si dà il via alla scrittura di questo dato nell'indirizzo ancora memorizzato all'interno di MAR. Alla fine si fa un salto incondizionato al main.

Il programma che permette di testare questa modifica è riportato qui sotto.

```
.main
.var
a
.endvar
BIPUSH 0x1
BIPUSH 0x0
IAND
ISTORE a
```

```
HALT
.endmethod
```

---

È necessario verificare che la variabile "a" contenga 1 anziché 0. Il process "wavegen\_proc" deve quindi essere scritto nel seguente modo.

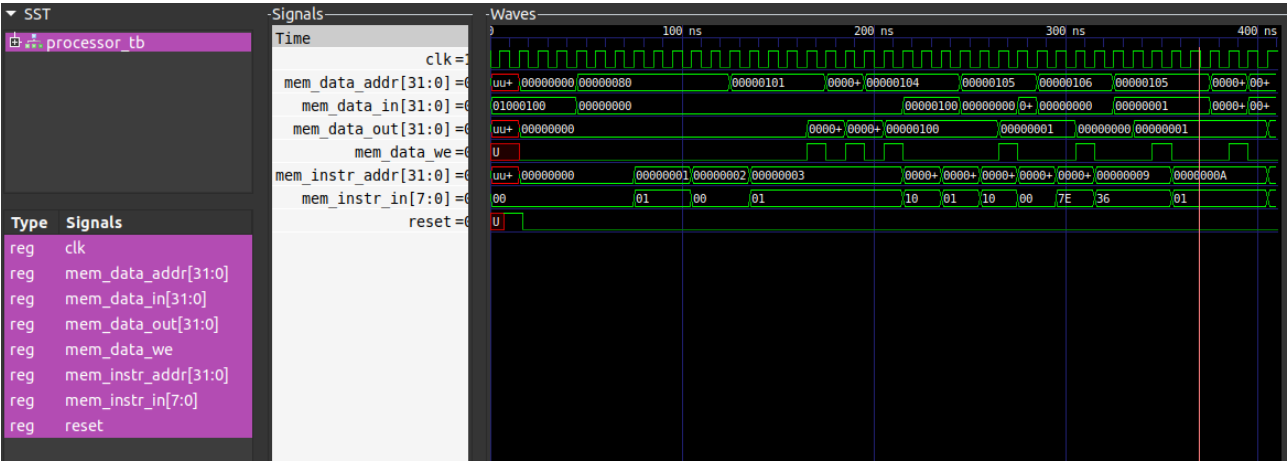
---

```
-- Waveform generation
wavegen_proc: process
begin
    wait until clk = '1';
    wait for 2 ns;
    reset <= '1';
    wait for 10 ns;
    reset <= '0';
    wait until mem_instr_addr = x"0000000A" and mem_data_we = '1';
    assert mem_data_out = x"00000001" report "Bad calculated value" severity failure;
    wait until mem_instr_addr = x"0000000B";
end_run := true;
    wait;
end process wavegen_proc;
```

---

```
Federica@federica-VirtualBox:~/esercitazione_mic/amic-0$ cmake --build build --target
create_control_store
Built target assemble_microprogram
Built target create_control_store
Federica@federica-VirtualBox:~/esercitazione_mic/amic-0$ cmake --build build --target
create_ram
Built target assemble_program
Built target create_ram
Federica@federica-VirtualBox:~/esercitazione_mic/amic-0$ cmake --build build --target
check
Built target index
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/common_defs.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/control_store.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/control_unit.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/alu.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/datapath.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/processor.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/main/vhdl/dp_ar_ram.vhd
analyze /home/federica/esercitazione_mic/amic-0/src/test/vhdl/processor_tb.vhd
elaborate processor_tb
Built target src.test.vhdl.processor_tb
analyze /home/federica/esercitazione_mic/amic-0/src/test/vhdl/alu_tb.vhd
elaborate alu_tb
Built target src.test.vhdl.alu_tb
analyze /home/federica/esercitazione_mic/amic-0/src/test/vhdl/control_unit_tb.vhd
elaborate control_unit_tb
Built target src.test.vhdl.control_unit_tb
analyze /home/federica/esercitazione_mic/amic-0/src/test/vhdl/datapath_tb.vhd
elaborate datapath_tb
Built target src.test.vhdl.datapath_tb
Test project /home/federica/esercitazione_mic/amic-0/build
  Start 1: src.test.vhdl.alu_tb
1/4 Test #1: src.test.vhdl.alu_tb ..... Passed    0.01 sec
  Start 2: src.test.vhdl.control_unit_tb
2/4 Test #2: src.test.vhdl.control_unit_tb .... Passed    0.01 sec
  Start 3: src.test.vhdl.datapath_tb
3/4 Test #3: src.test.vhdl.datapath_tb ..... Passed    0.01 sec
  Start 4: src.test.vhdl.processor_tb
4/4 Test #4: src.test.vhdl.processor_tb ..... Passed    0.02 sec

100% tests passed, 0 tests failed out of 4
```



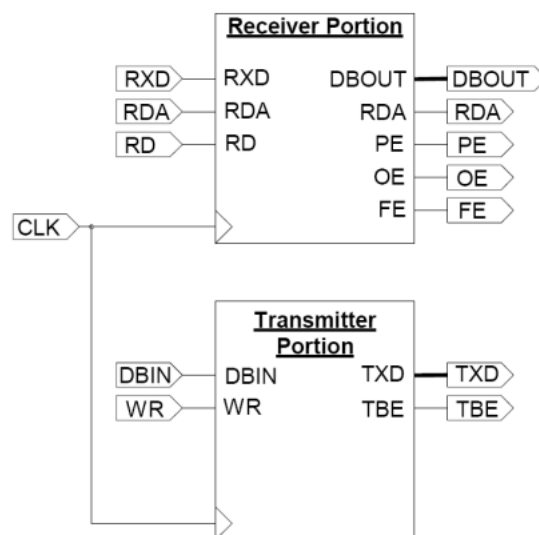
## Capitolo 6: Interfaccia Seriale

### Esercizio 10: RS232

#### Progetto e architettura

Il progetto prevede un sistema composto da due unità (che condividono lo stesso segnale di clock) comunicanti tramite interfaccia seriale, tale che, dato un segnale **start**, venga prelevato un byte dalla ROM del sistema A e inviato al sistema B per essere scritto in una memoria.

Il dispositivo utilizzato per la comunicazione seriale asincrona è la **UART** (Universal Asynchronous Receiver Transmitter) nel quale la comunicazione avviene secondo un preciso protocollo. Lo standard impiegato è l'**RS-232** (Recommended Standard 232) che è uno standard di interfaccia seriale che specifica la connessione elettrica, i segnali e le regole per la trasmissione seriale dei dati. Per quanto riguarda le specifiche elettriche, l'intervallo di tensione [+3 V, +15 V] è usato per rappresentare il valore logico basso e l'intervallo [-3 V, -15 V] per il valore logico alto; questa soluzione permette di tollerare meglio il rumore.



Dal punto di vista architetturale la componente è formata da una porzione dedicata alla ricezione e una alla trasmissione, ognuna delle quali contiene:

- Un registro per la serializzazione/deserializzazione dei bit inviati/ricevuti (PISO per il trasmettitore e SIPO per il ricevitore).
- Un contatore per scandire i bit del frame.
- Un contatore per gestire l'invio dei singoli bit in base al baud rate scelto.
- Un'unità di controllo.

I segnali di ingresso e uscita del componente RS232 sono:

- **TXD** per la trasmissione dei dati (di default ha valore 1)
- **RXD** per la ricezione dei dati
- **CLK** (Clock)

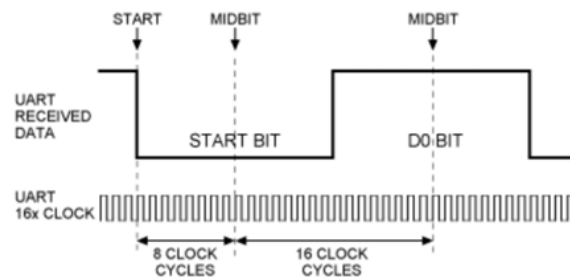


- **DBIN** (Data Bus In) contiene l'intero vettore di bit che deve essere trasmesso
- **DBOUT** (Data Bus Out) contiene il vettore di bit ricevuto nella sua interezza
- **RDA** (Read Data Available) informa che si è conclusa la ricezione dei dati
- **TBE** (Transfer Bus Empty) informa che si è pronti a inviare nuovi dati
- **RD** (Read Strobe) avvisa il ricevitore di leggere i dati (se RD = 1 allora RDA = 0)
- **WR** (Write Strobe) avvisa il trasmettitore di inviare i dati (se WR = 1 allora TBE = 0)
- **PE** (Parity Error Flag)
- **FE** (Frame Error Flag)
- **OE** (Overwrite Error Flag)
- **RST** (Reset)

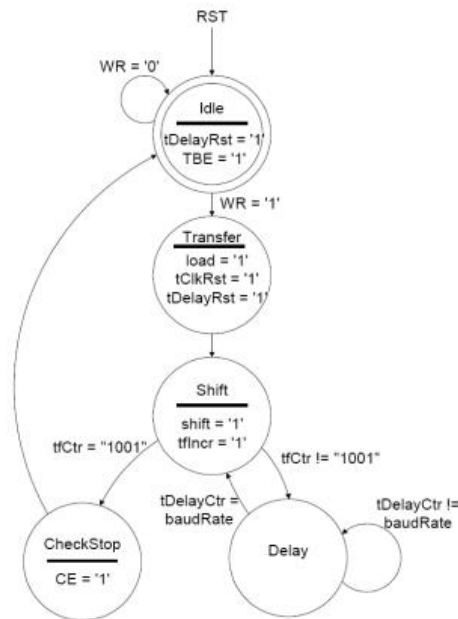
Il componente usato usa un baud rate, una velocità, di 9600, ovvero trasmette e riceve 9600 bit al secondo. La frequenza di campionamento dell'interconnessione deve però essere diversa tra le porzioni di trasmissione e ricezione, dove il ricevitore deve presentare una frequenza 16 o 64 volte maggiore.

Il codice VHDL contenuto in "RS232RefComp.vhd" genera, a partire da CLK, un clock in ricezione (**rCLK**) con duty cycle del 50% e da quest'ultimo viene ricavato il clock di trasmissione (**tCLK**). rCLK è ottenuto dal segnale **clkDiv** nel seguente modo: clkDiv è incrementato ogni colpo di clock (CLK), quando clkDiv raggiunge il valore massimo **baudDivide**, ossia 163, si azzerà e viene invertito rClk. Inoltre, ogni volta che si osserva un fronte di salita di rClk, il segnale **rClkDiv**, a quattro bit, viene incrementato. tClk non è altro che l'ultimo bit di **rClkDiv** e quindi esso commuta ogni otto colpi di rClk.

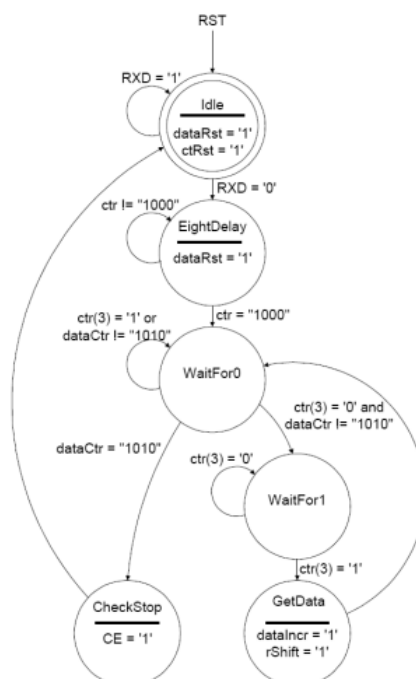
In questo modo si ottiene una frequenza di clock per il ricevitore 16 volte quella del trasmettitore. Appena viene riconosciuto lo start bit, il ricevitore si posiziona al "centro" del bit contando un certo numero di fronti del suo clock, in questo caso 8; successivamente campionerà il filo collegato a RXD ogni 16 colpi così da posizionarsi sempre nel mezzo.



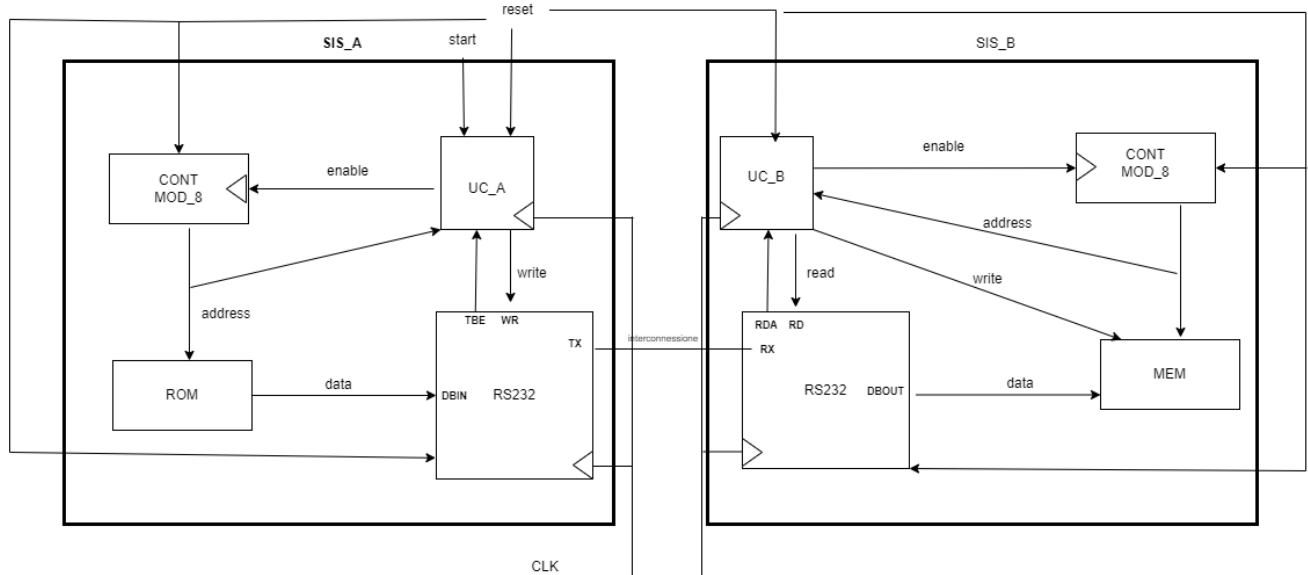
Di seguito è riportato il diagramma degli stati del controller del "Transmitter Portion" del componente UART. Partendo da **IDLE** si rimane in questo stato finché non si alza il segnale esterno **WR**, che indica l'inizio della trasmissione. Si passa, quindi, allo stato successivo di **Transfer** in cui si alza sia **load** (viene caricata la stringa nello Shift Register) che **tClkRst** (viene resettato il contatore di bit inviati). Nella fase successiva di **Shift**, il bit uscente dallo shift register viene trasmesso. Si transita nello stato di **Delay**, dove si rimane per un certo intervallo di tempo in accordo con il bound rate per poi ritornare in Shift. Questo ciclo si ripete finché il frame non è stato completamente trasmesso ovvero quando **tfCtr**, il quale conta quanti bit sono stati inviati, è uguale a 12 (11 bit del frame più 1 per separare la trasmissione di due frame successivi); a questo punto di transizione nello stato di **IDLE**.



Successivamente è illustrato il diagramma a stati dell'unità di controllo del RX del componente UART. A partire dallo stato **IDLE**, si fa una transizione allo stato successivo se RXD si abbassa, evento che indica l'inizio di una comunicazione. Si rimane nello stato **EightDelay** il tempo necessario per contare 8 impulsi, in modo tale da posizionarsi al centro del bit di start. Appena **ctr** arriva ad 8, si passa allo stato successivo **WaitFor0**; gli stati WaitFor0 e **WaitFor1** sono strutturati in maniera tale che la lettura di RXD avviene sempre al centro della trasmissione. In altre parole, il loro funzionamento combinato permette di contare 16 colpi di clock: partendo da WaitFor0 quando **ctr(3)=0** si sono contati 8 impulsi, si va quindi nello stato **WaitFor1** e qui si rimane finchè **ctr(3)=1** ovvero si sono contati altri 8 impulsi, alla fine si fa la transizione in **GetData**. Nello stato GetData si incrementa **dataCtr**, il contatore dei bit di dato ricevuti, si fornisce il segnale di shift e si torna in WaitFor0. Si ripete in loop fino a quando **dataCtr** non diventa 10 (8 bit di dati, 1 di parità e 1 di stop), a questo punto ci si sposta nello stato **CheckStop** che abilita il controllo degli errori alzando il bit CE. Infine si transiziona di nuovo nello stato di riposo.



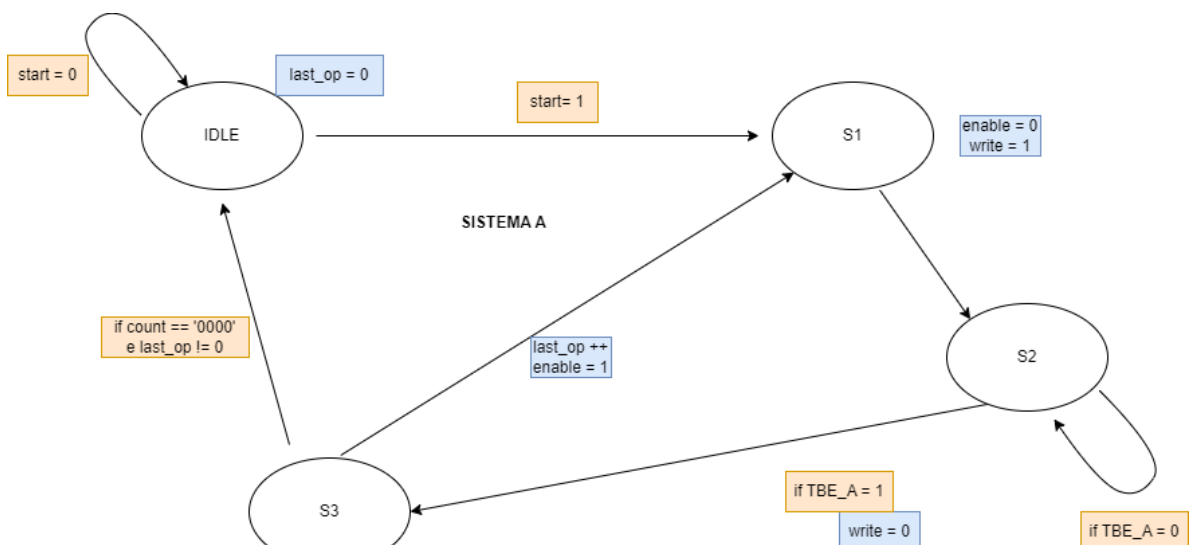
Di seguito è disegnata l'architettura dei due sistemi comunicanti. Si osservi che il segnale di tempificazione, il clock, è condiviso.



Il sistema A è composto da un'unità di controllo, una componente UART RS232 con TX connesso a RX del ricevitore, un contatore modulo 8 e una memoria ROM. Un segnale di start fornito dall'esterno al sistema dà il via alla trasmissione seriale, a intervalli regolari, dei dati contenuti nella ROM (l'indirizzo è fornito dal contatore) al sistema B, tramite UART.

Il diagramma degli stati finiti del sistema A è descritto successivamente.

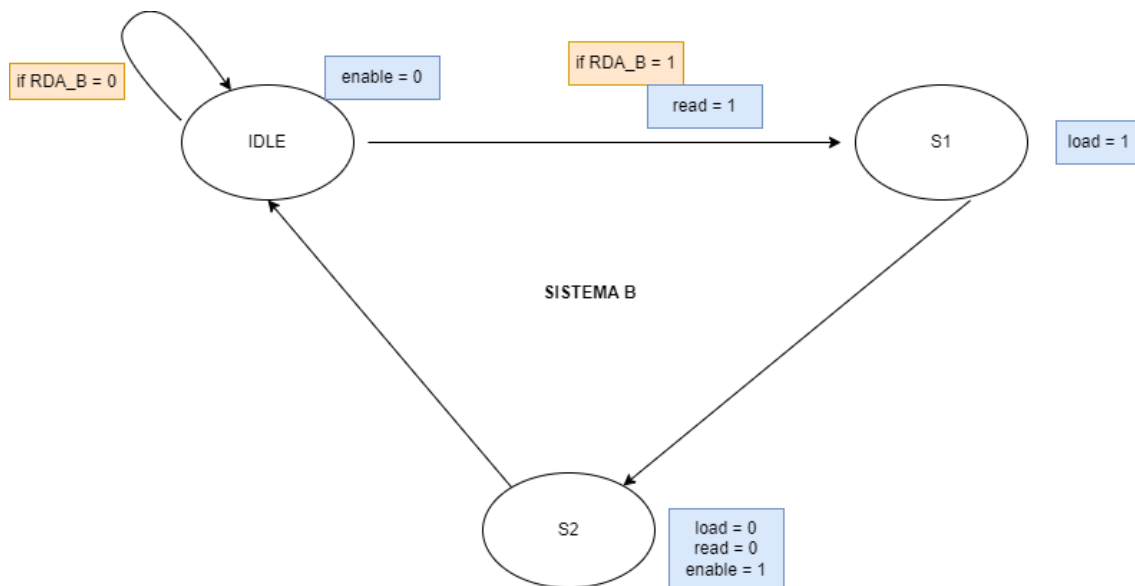
Per determinare lo stato in un certo istante l'unità di controllo riceve dal componente RS232 il segnale TBE che ha di default un valore logico alto. Nel momento in cui WR si alza e si inizia una trasmissione, TBE di conseguenza si abbassa. Solo nel momento in cui è finita una trasmissione TBE è di nuovo 1 quindi si torna allo stato di partenza.



Si parte da uno stato iniziale **IDLE**. Se viene fornito dall'esterno un segnale start si passa a uno stato **S1**, dove **enable** è abbassato e **write** è impostato ad 1 (write va in ingresso al componente RS232 come segnale WR

che dà inizio alla trasmissione). Successivamente si passa ad uno stato **S2** in cui si permane finché il segnale **TBE\_A**, che l'unità di controllo riceve dall'UART, si alza; in questo caso viene abbassato write. Se il contatore mod-8 ha raggiunto il valore massimo, si azzerà e si ritorna allo stato IDLE, altrimenti si abilita **enable** per incrementare il contatore e si transiziona nello stato **S1**.

Il sistema B è composto, anch'esso da un'unità di controllo, un componente RS232 con **RX** connesso al **TX** del trasmettitore, un contatore mod-8 e una memoria. Il diagramma a stati dell'unità di controllo di B sfrutta il segnale **RDA** in uscita al componente RS232 per determinare quando un frame è stato completamente ricevuto.



Partendo da uno stato **IDLE**, il sistema permane qui fino a quando **RDA\_B** non assume valore logico alto. Si fa, quindi, una transizione allo stato **S1** in cui si abilita il segnale **read** per la lettura del dato, che andrà in ingresso al componente RS232 come RD, e si alza **load** per il caricamento dei dati ricevuti nella memoria. Successivamente nello stato **S3** vengono abbassati i segnali di **load** e **read**, mentre l'**enable** del contatore viene abilitato. Infine, si torna ad **IDLE**.

### Implementazione

Per l'unità di controllo del sistema A il diagramma degli stati finiti è stato implementato mediante due process. Invece, per l'unità di controllo di B è stata utilizzato un unico process.

---

#### sistema\_complessivo.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sistema_complessivo is
    Port (
        clock, start, reset : in std_logic
    );
end sistema_complessivo;
  
```

```

architecture Behavioral of sistema_complessivo is
    component sistema_A is
        Port (clock, start, reset: in std_logic;
              interconnessione : out std_logic
              );
    end component;

    component sistema_B is
        Port (clock, reset, interconnessione : in std_logic );
    end component;

    signal interconnessione : STD_LOGIC;
begin
    sis_A : sistema_A port map (clock => clock, start => start, reset =>
reset, interconnessione => interconnessione);

    sis_B : sistema_B port map (clock => clock, reset => reset,
interconnessione => interconnessione);
end Behavioral;

```

---

#### sistema\_A.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sistema_A is
    Port (clock, start, reset: in std_logic;
          interconnessione : out std_logic
          );
end sistema_A;

architecture Behavioral of sistema_A is

    component ROM is port(
        address : in  std_logic_vector(3 downto 0);
        d_out    : out std_logic_vector(7 downto 0)
    );
    end component;

    component cont_mod_8 is
    generic (
        MAX_VALUE : integer := 7
    );
    port (
        clock : in std_logic;
        reset : in std_logic;
        count : out std_logic_vector(3 downto 0)
    );

```

```

end component;

component Rs232RefComp is
Port (
    TXD      : out std_logic      := '1';
    RXD      : in  std_logic;
    CLK      : in  std_logic;      --Master Clock
    DBIN     : in  std_logic_vector (7 downto 0); --Data Bus in
    DBOUT    : out std_logic_vector (7 downto 0); --Data Bus out
    RDA      : inout std_logic;    --Read Data Available(1
quando il dato è disponibile nel registro rdReg)
    TBE      : inout std_logic     := '1';      --Transfer Bus Empty(1
quando il dato da inviare è stato caricato nello shift register)
    RD       : in  std_logic;      --Read Strobe(se 1
significa "leggi" --> fa abbassare RDA)
    WR       : in  std_logic;      --Write Strobe(se 1
significa "scrivi" --> fa abbassare TBE)
    PE       : out std_logic;      --Parity Error Flag
    FE       : out std_logic;      --Frame Error Flag
    OE       : out std_logic;      --Overwrite Error Flag
    RST      : in  std_logic := '0');      --Master Reset
end component;

component control_unit_A is
Port ( clock : in STD_LOGIC;
      start  : in STD_LOGIC;
      reset  : in STD_LOGIC;
      write  : out STD_LOGIC;
      enable : out STD_LOGIC;
      count  : in STD_LOGIC_VECTOR(3 downto 0);
      TBE_A  : inout STD_LOGIC
    );
end component;

signal address : std_logic_vector(3 downto 0) := (others=>'0');
signal write, -- della UART
      TBE_A,
      enable : std_logic; -- del contatore
signal data, DBIN, DBOUT : std_logic_vector(7 downto 0); -- DBOUT non
usato
signal RDA, PE, FE, OE : std_logic := '0'; -- non usati

begin
    mem_rom : ROM port map (address => address, d_out => data);

    cont_A : cont_mod_8 port map(clock => enable, reset => reset, count =>
address);

    contr_unit_A : control_unit_A port map(clock => clock,
      start => start,

```

```

        reset => reset,
        write => write,
        enable => enable,
        count => address,
        TBE_A => TBE_A);

UART_A : Rs232RefComp port map(
    TXD => interconnessione,
    RXD => '0',
    CLK => clock,
    DBIN => data,
    DBOUT => DBOUT,
    RDA => RDA,
    TBE => TBE_A,
    RD => '0',
    WR => write,
    PE => PE,
    FE => FE,
    OE => OE,
    RST => reset);

end Behavioral;

```

---

## control\_unit\_A.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity control_unit_A is
    Port ( clock : in STD_LOGIC;
          start : in STD_LOGIC;
          reset : in STD_LOGIC;
          write : out STD_LOGIC;
          enable : out STD_LOGIC;
          count : in STD_LOGIC_VECTOR(3 downto 0);
          TBE_A : in STD_LOGIC
    );
end control_unit_A;

architecture Behavioral of control_unit_A is

    type state is (idle, s1, s2, s3);
    signal current_state, next_state: state;

begin
    reg_stato: process(clock)
    begin
        if(clock'event and clock='1') then

```

```

        if(reset='1') then
            current_state <= idle;
        else
            current_state <= next_state;
        end if;
    end if;
end process;

comb: process(current_state, start, count, TBE_A)
variable last_op : integer :=0;
begin
    case current_state is

        when idle =>
            last_op := 0;
            if(start='1') then
                next_state <= s1;
            else
                next_state <= idle;
            end if;

        when s1 =>
            enable <= '0';
            write <= '1';
            next_state <= s2;

        when s2 =>

            if(TBE_A='0') then
                next_state <= s2;
            else
                write <= '0';
                next_state <= s3;
            end if;

        when s3 =>
            if(count="0000" and last_op/=0) then
                next_state <= idle;
            else
                last_op := 1;
                enable <= '1';
                next_state <= s1;
            end if;

        end case;
    end process;

end Behavioral;

```

---



## sistema\_B.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sistema_B is
    Port (clock, reset, interconnessione : in std_logic);
end sistema_B;

architecture Behavioral of sistema_B is

    component MEM is
        Port (
            address : in STD_LOGIC_VECTOR(3 downto 0);
            data_in : in STD_LOGIC_VECTOR(7 downto 0);
            write : in STD_LOGIC;
            read : in STD_LOGIC;
            data_out : out STD_LOGIC_VECTOR(7 downto 0)
        );
    end component;

    component cont_mod_8 is
        generic (
            MAX_VALUE : integer := 7
        );
        port (
            clock : in std_logic;
            reset : in std_logic;
            count : out std_logic_vector(3 downto 0)
        );
    end component;

    component control_unit_B is
        Port ( clock : in STD_LOGIC;
            reset : in STD_LOGIC;
            enable : out STD_LOGIC;
            load : out STD_LOGIC;
            read : out STD_LOGIC;
            RDA_B : in STD_LOGIC
        );
    end component;

    component Rs232RefComp is
        Port (
            TXD      : out std_logic      := '1';
            RXD      : in  std_logic;
            CLK      : in  std_logic;
            DBIN     : in  std_logic_vector (7 downto 0); --Master Clock
            DBOUT    : out std_logic_vector (7 downto 0); --Data Bus in
            --Data Bus out
        );
    end component;

end architecture Behavioral of sistema_B;
```

```

        RDA : inout std_logic;           --Read Data Available(1
quando il dato è disponibile nel registro rdReg)
        TBE : inout std_logic := '1';    --Transfer Bus Empty(1
quando il dato da inviare è stato caricato nello shift register)
        RD   : in  std_logic;            --Read Strobe(se 1
significa "leggi" --> fa abbassare RDA)
        WR   : in  std_logic;            --Write Strobe(se 1
significa "scrivi" --> fa abbassare TBE)
        PE   : out std_logic;            --Parity Error Flag
        FE   : out std_logic;            --Frame Error Flag
        OE   : out std_logic;            --Overwrite Error Flag
        RST   : in  std_logic := '0');    --Master Reset
    end component;

    signal data, data_out : std_logic_vector(7 downto 0); -- data_out non
utilizzato
    signal address : std_logic_vector(3 downto 0);
    signal enable, -- del contatore
        load, -- della memoria
        read: std_logic; -- della UART
    signal TXD, TBE, PE, FE, OE : std_logic; -- non usati
    signal DBIN, DBOUT : STD_LOGIC_VECTOR(7 DOWNTO 0); -- DBIN non usati
    signal RDA_B : std_logic := '0';

begin
    mem_data : MEM port map(address => address, data_in => data, write =>
load, read => '0', data_out => data_out);

    cont_B : cont_mod_8 port map(clock => enable, reset => reset, count =>
address);

    contr_unit_B : control_unit_B port map(clock => clock, reset => reset,
read => read, enable => enable, load => load, RDA_B => RDA_B);

    UART_B : Rs232RefComp port map(
        TXD => TXD,
        RXD => interconnessione,
        CLK => clock,
        DBIN => "00000000",
        DBOUT => data,
        RDA => RDA_B,
        TBE => TBE,
        RD => read,
        WR => '0',
        PE => PE,
        FE => FE,
        OE => OE,
        RST => reset);

end Behavioral;

```

---

## control\_unit\_B.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity control_unit_B is
    Port ( clock : in STD_LOGIC;
          reset : in STD_LOGIC;
          enable : out STD_LOGIC;
          load : out STD_LOGIC;
          read : out STD_LOGIC;
          RDA_B : in STD_LOGIC
        );
end control_unit_B;

architecture Behavioral of control_unit_B is

    type state is (idle, s1, s2);
    signal current_state: state;

begin

    process(clock)
    begin
        if rising_edge(clock) then
            if reset = '1' then
                current_state <= idle;
            else
                case current_state is
                    when idle =>
                        enable <= '0';
                        if RDA_B = '1' then
                            read <= '1';
                            current_state <= s1;
                        else
                            current_state <= idle;
                        end if;
                    when s1 =>
                        load <= '1';
                        current_state <= s2;
                    when s2 =>
                        load <= '0';
                        read <= '0';
                        enable <= '1';
                        current_state <= idle;
                    end case;
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```

```

        end case;
    end if;
end if;
end process;

end Behavioral;

```

---

## cont\_mod\_8.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use ieee.numeric_std.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity cont_mod_8 is
    generic (
        MAX_VALUE : integer := 7
    );
    port (
        clock : in std_logic;
        reset : in std_logic;
        count : out std_logic_vector(3 downto 0) -- std_logic_vector di 4 bit
    );
end entity cont_mod_8;

architecture Behavioral of cont_mod_8 is
    signal counter : integer range 0 to MAX_VALUE := 0;
begin
    process(clock, reset)
    begin
        if reset = '1' then
            counter <= 0;
        elsif rising_edge(clock) then
            if counter < MAX_VALUE then
                counter <= counter + 1;
            else
                counter <= 0;
            end if;
        end if;
    end process;

    count <= std_logic_vector(to_unsigned(counter, count'length));
end architecture Behavioral;

```

---

## ROM.vhd

```

library ieee;
use ieee.std_logic_1164.all;

```

```

use ieee.numeric_std.all;

entity ROM is port(
    address : in std_logic_vector(3 downto 0);
    d_out    : out std_logic_vector(7 downto 0)
);
end entity ROM;

architecture RTL of ROM is
    type MEMORY_16_8 is array (0 to 15) of std_logic_vector(7 downto 0);

    constant ROM_16_8 : MEMORY_16_8 := (
        "00000000",
        "00000001",
        "00000010",
        "00000011",
        "00000100",
        "00000101",
        "00000110",
        "00000111",
        "00001000",
        "00001001",
        "00001010",
        "00001011",
        "00001100",
        "00001101",
        "00001110",
        "00001111"
    );

begin
    main : process(address)
    begin
        d_out <= ROM_16_8(to_integer(unsigned(address)));
    end process main;
end architecture RTL;

```

---

## MEM.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity MEM is
    Port (
        address : in STD_LOGIC_VECTOR(3 downto 0);
        data_in  : in STD_LOGIC_VECTOR(7 downto 0);
        write    : in STD_LOGIC;
    );
end entity MEM;

```

```

        read : in STD_LOGIC;
        data_out : out STD_LOGIC_VECTOR(7 downto 0)
    );
end MEM;

architecture Behavioral of MEM is

    type Memory_Array is array (0 to 7) of STD_LOGIC_VECTOR(7 downto 0);
    signal mem : Memory_Array := (others => "00000000");

begin

    process(address, data_in, write, read)
    begin
        if write = '1' then
            mem(conv_integer(address)) <= data_in;
        elsif read = '1' then
            data_out <= mem(conv_integer(address));
        else
            data_out <= (others => '0');
        end if;
    end process;
end Behavioral;

```

---

*Simulazione*

---

### **tb\_sistema\_compressivo.vhd**

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity tb_sistema_compressivo is
end entity tb_sistema_compressivo;

architecture Behavioral of tb_sistema_compressivo is

    signal clock : std_logic := '0';
    signal start : std_logic := '0';
    signal reset : std_logic := '0';

begin

    sis_tot: entity work.sistema_compressivo
    port map (
        clock => clock,
        start => start,
        reset => reset
    );

```

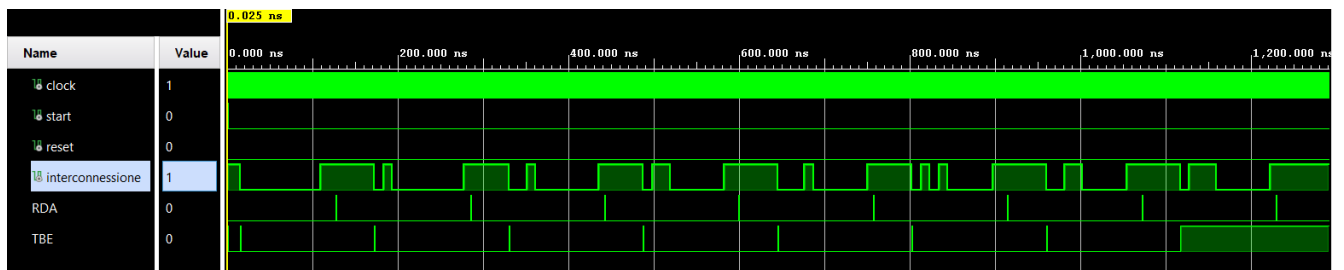
```

clock <= not clock after 1 ps;

stimulus: process
begin
    wait for 2 ps;
    start <= '1';
    wait for 2 ps;
    start <= '0';

    wait;
end process stimulus;
end architecture Behavioral;

```



## Capitolo 7: Switch Multistadio

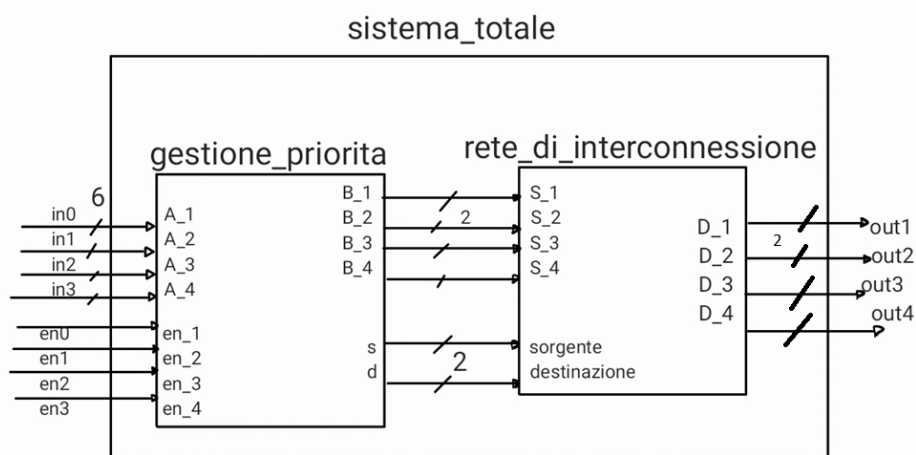
### Esercizio 11: Switch Multistadio

#### Progetto e architettura

Il circuito digitale complessivo è formato da due componenti principali: **gestione\_priorità** e **rete\_di\_interconnessione**.

Il sistema ha quattro ingressi di abilitazione da un bit (**en0**, **en1**, **en2**, **en3**) e quattro ingressi di dati da 6 bit (**in0**, **in1**, **in2**, **in3**). Inoltre, ci sono quattro uscite di dati (**out0**, **out1**, **out2**, **out3**) di larghezza 2 bit ciascuna.

L'architettura del sistema, ossia come i vari blocchi sono collegati alle porte di ingresso e uscita e interconnessi tra loro attraverso segnali interni, è illustrata nel diagramma sottostante.



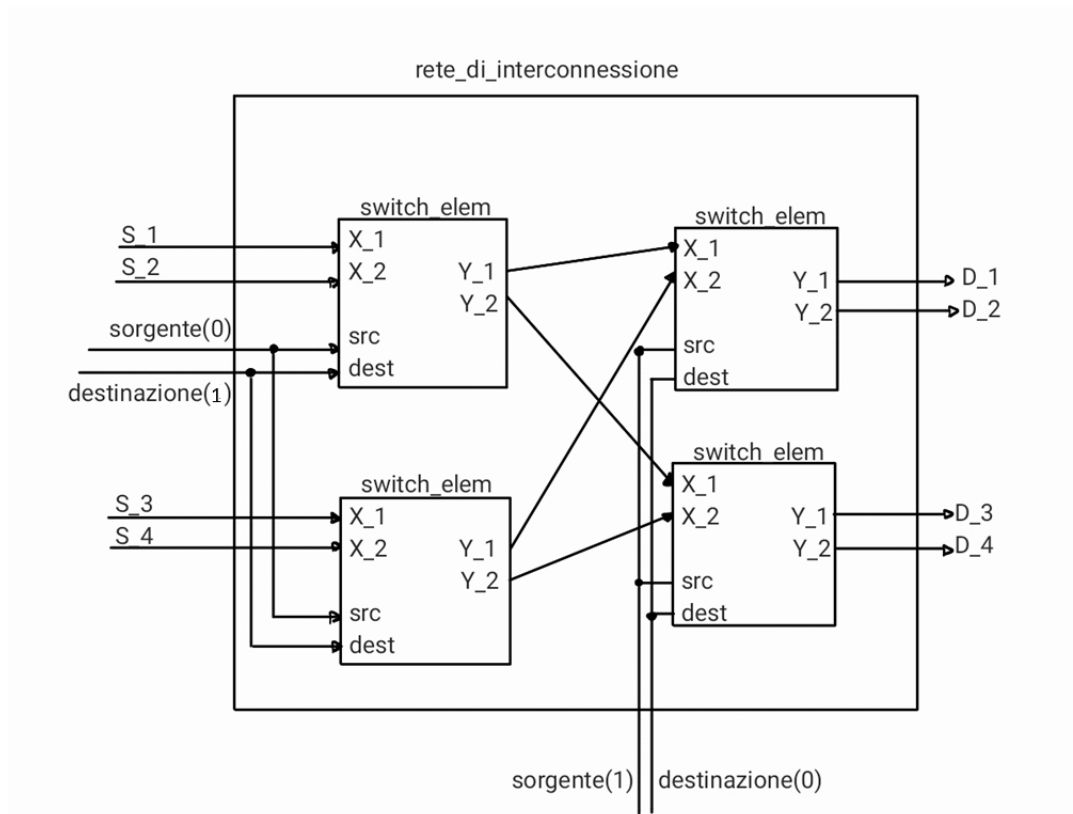
Il modulo **gestione\_priorita** è stato progettato per selezionare la “trasmissione” con la priorità più alta tra quattro possibilità (**A\_1**, **A\_2**, **A\_3**, **A\_4**). I primi due bit di ciascun ingresso indicano la sorgente della trasmissione, i successivi due la destinazione, e gli ultimi due contengono i dati da trasmettere. In base ai quattro segnali di abilitazione in ingresso (**en\_1**, **en\_2**, **en\_3**, **en\_4**), il modulo “seleziona” uno tra gli input disponibili definendo i valori degli output. Le uscite di **gestione\_priorita** sono i segnali di sorgente (**s**) e destinazione (**d**) e i quattro segnali di dati (**B\_1**, **B\_2**, **B\_3**, **B\_4**), tutti da due bit, che sono in entrata alla rete di interconnessione.

In particolare, all’interno dell’architettura “Behavioral” viene creato il segnale “**k**” che rappresenta la priorità decrescente. I segnali **s**, **d**, **B\_1**, **B\_2**, **B\_3** e **B\_4** sono definiti proprio in base a questa **k**. In dettaglio, il valore di **k** è “00” quando è attivo il segnale di abilitazione **en\_1**. Nel caso in cui **en\_1** sia basso, **k** è “01” se è attivo **en\_2**. Se sia **en\_1** che **en\_2** sono bassi, **k** assume il valore “10” se è attivo **en\_3**, altrimenti è uguale a “11” se è attivo **en\_4**. Infine, se nessuna delle condizioni precedenti è verificata, il valore di **k** è impostato su “--” (don't care). Assegnazioni di questo tipo sono rese possibili da costruito **when...else** (assegnazione concorrente condizionale).

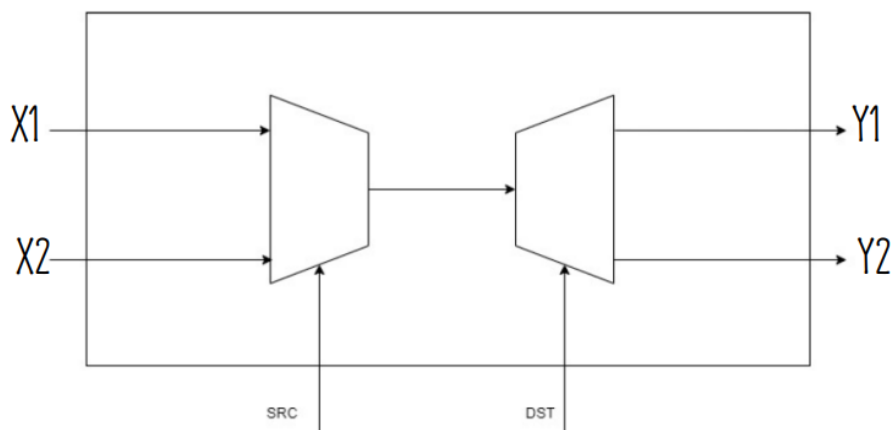
La rete di interconnessione non è altro che una “omega network”. Questa rete multistadio è composta da quattro switch elementari (**switch\_elem**) opportunamente collegati tra di loro, ciascuno avente due ingressi (**X\_1**, **X\_2**) e due uscite (**Y\_1**, **Y\_2**). L’obiettivo è quello di instradare un segnale a due bit dalla una delle quattro sorgenti (**S\_1**, **S\_2**, **S\_3**, **S\_4**) a una delle quattro destinazioni (**D\_1**, **D\_2**, **D\_3**, **D\_4**) in base ai valori di due segnali di selezione (**sorgente**, **destinazione**) di lunghezza 2.



I quattro switch sono collegati tra di loro e ai porti di I/O della rete in modo conforme all'algoritmo di “**perfect shuffling**”, ovvero le interconnessioni risultanti prevedono due livelli di switch e seguono uno schema specifico che alterna gli indirizzi sorgente/destinazione. In altre parole, gli ingressi vengono instradati attraverso due switch iniziali, i cui risultati intermedi vengono poi opportunamente inoltrati attraverso altri due switch per raggiungere le uscite desiderate. La specifica disposizione delle connessioni tra gli switch elementari può essere osservata nell'immagine successiva.



Lo switch elementare (switch\_elem) è realizzato combinando un multiplexer 2 a 1 (**mux\_2\_1**) e un demultiplexer 1 a 2 (**demux\_1\_2**) in serie. L'entity definisce quattro ingressi (**X\_1**, **X\_2**, **src**, **dest**) e due uscite (**Y\_1**, **Y\_2**): X\_1 e X\_2 sono i bit provenienti da due possibili sorgenti, src è il segnale di selezione della sorgente, dest della destinazione e Y\_1 e Y\_2 sono singoli segnali connessi alle due potenziali destinazioni. Il segnale interno mux\_out\_demux\_in mette in collegamento l'uscita del multiplexer con l'ingresso del demultiplexer. La struttura completa di switch\_elem è illustrata di seguito.



## **mux\_2\_1.vhd**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux_2_1 is
    Port (d_in_1 : in std_logic_vector(1 downto 0);
          d_in_2 : in std_logic_vector(1 downto 0);
          sel : in std_logic;
          d_out : out std_logic_vector(1 downto 0));
end mux_2_1;

architecture Dataflow of mux_2_1 is
begin
    d_out <= d_in_1 when sel='0' else
              d_in_2 when sel='1' else
              "--";

end Dataflow;
```

---

## **demux\_1\_2.vhd**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity demux_1_2 is
    Port (d_in : in std_logic_vector(1 downto 0);
          sel : in std_logic;
          d_out_1 : out std_logic_vector(1 downto 0);
          d_out_2 : out std_logic_vector(1 downto 0));
end demux_1_2;

architecture Dataflow of demux_1_2 is
begin
    d_out_1 <= d_in when sel='0' else "--";

    d_out_2 <= d_in when sel='1' else "--";

end Dataflow;
```

---

## **switch\_elem.vhd**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity switch_elem is
    Port (X_1, X_2 : in std_logic_vector(1 downto 0);
          src, dest : in std_logic;
```

```

        Y_1, Y_2 : out std_logic_vector(1 downto 0));
end switch_elem;

architecture Structural of switch_elem is

    component mux_2_1 is
        Port (d_in_1 : in std_logic_vector(1 downto 0);
              d_in_2 : in std_logic_vector(1 downto 0);
              sel : in std_logic;
              d_out : out std_logic_vector(1 downto 0));
    end component;

    component demux_1_2 is
        Port (d_in : in std_logic_vector(1 downto 0);
              sel : in std_logic;
              d_out_1 : out std_logic_vector(1 downto 0);
              d_out_2 : out std_logic_vector(1 downto 0));
    end component;

    signal mux_out_demux_in : std_logic_vector(1 downto 0);

begin
    multiplexer_2_1 : mux_2_1 port map(d_in_1 => X_1, d_in_2 => X_2, sel =>
src, d_out => mux_out_demux_in);

    demultiplexer_1_2 : demux_1_2 port map(d_in => mux_out_demux_in, sel =>
dest, d_out_1 => Y_1, d_out_2 => Y_2);
end Structural;

```

---

## gestione\_priorita.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity gestione_priorita is
    Port (A_1, A_2, A_3, A_4 : in std_logic_vector(5 downto 0); -- i primi
due bit determinano la sorgente, successivi due la destinazione, gli ultimi
due i dati da trasmettere
          en_1, en_2, en_3, en_4 : in std_logic;
          s, d : out std_logic_vector(1 downto 0);
          B_1, B_2, B_3, B_4 : out std_logic_vector(1 downto 0));
end gestione_priorita;

architecture Behavioral of gestione_priorita is

    signal k : std_logic_vector(1 downto 0); -- priorit 
    signal temp : std_logic_vector(1 downto 0);

begin

```

```

k <= "00" when en_1='1' else
    "01" when en_2='1' else
    "10" when en_3='1' else
    "11" when en_4='1' else
    "--";

s <= A_1 (5 downto 4) when k ="00" else
    A_2 (5 downto 4) when k ="01" else
    A_3 (5 downto 4) when k ="10" else
    A_4 (5 downto 4) when k ="11" else
    "--";

d <= A_1 (3 downto 2) when k ="00" else
    A_2 (3 downto 2) when k ="01" else
    A_3 (3 downto 2) when k ="10" else
    A_4 (3 downto 2) when k ="11" else
    "--";

temp <= A_1(1 downto 0) when k="00" else
    A_2(1 downto 0) when k="01" else
    A_3(1 downto 0) when k="10" else
    A_4(1 downto 0) when k="11" else
    "--";

B_1 <= temp when k = "00" else "--";
B_2 <= temp when k = "01" else "--";
B_3 <= temp when k = "10" else "--";
B_4 <= temp when k = "11" else "--";
end Behavioral;

```

---

## rete\_di\_interconnessione.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity rete_di_interconnessione is -- omega network
    Port (S_1, S_2, S_3, S_4 : in std_logic_vector(1 downto 0);
          D_1, D_2, D_3, D_4 : out std_logic_vector(1 downto 0);
          sorgente : in std_logic_vector(1 downto 0);
          destinazione : in std_logic_vector(1 downto 0));
end rete_di_interconnessione;

architecture Behavioral of rete_di_interconnessione is
    component switch_elem is
        Port (X_1, X_2 : in std_logic_vector(1 downto 0);
              src, dest : in std_logic;
              Y_1, Y_2 : out std_logic_vector(1 downto 0));
    end component;

```

```

    signal temp_1, temp_2, temp_3, temp_4 : std_logic_vector(1 downto 0);
begin
    switch_1 : switch_elem port map (
        X_1 => S_1,
        X_2 => S_2,
        src => sorgente(0),
        dest => destinazione(1),
        Y_1 => temp_1,
        Y_2 => temp_2
    );

    switch_2 : switch_elem port map (
        X_1 => S_3,
        X_2 => S_4,
        src => sorgente(0),
        dest => destinazione(1),
        Y_1 => temp_3,
        Y_2 => temp_4
    );

    switch_3 : switch_elem port map (
        X_1 => temp_1,
        X_2 => temp_3,
        src => sorgente(1),
        dest => destinazione(0),
        Y_1 => D_1,
        Y_2 => D_2
    );

    switch_4 : switch_elem port map (
        X_1 => temp_2,
        X_2 => temp_4,
        src => sorgente(1),
        dest => destinazione(0),
        Y_1 => D_3,
        Y_2 => D_4
    );

end Behavioral;

```

---

## sistema\_totale.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sistema_totale is
    Port (en0, en1, en2, en3 : in std_logic;
          in0, in1, in2, in3 : in std_logic_vector(5 downto 0));

```

```

        out0, out1, out2, out3 : out std_logic_vector(1 downto 0));
end sistema_totale;

architecture Behavioral of sistema_totale is
    component rete_di_interconnessione is
        Port (S_1, S_2, S_3, S_4 : in std_logic_vector(1 downto 0);
              D_1, D_2, D_3, D_4 : out std_logic_vector(1 downto 0);
              sorgente : in std_logic_vector(1 downto 0);
              destinazione : in std_logic_vector(1 downto 0));
    end component;

    component gestione_priorita is
        Port (A_1, A_2, A_3, A_4 : in std_logic_vector(5 downto 0);
              en_1, en_2, en_3, en_4 : in std_logic;
              s, d : out std_logic_vector(1 downto 0);
              B_1, B_2, B_3, B_4 : out std_logic_vector(1 downto 0));
    end component;

    signal temp1, temp2, temp3, temp4 : std_logic_vector(1 downto 0);
    signal source, destination : std_logic_vector(1 downto 0);
begin
    g_p : gestione_priorita port map (
        A_1 => in0,
        A_2 => in1,
        A_3 => in2,
        A_4 => in3,
        en_1 => en0,
        en_2 => en1,
        en_3 => en2,
        en_4 => en3,
        s => source,
        d => destination,
        B_1 => temp1,
        B_2 => temp2,
        B_3 => temp3,
        B_4 => temp4);

    r_d_i : rete_di_interconnessione port map (
        S_1 => temp1,
        S_2 => temp2,
        S_3 => temp3,
        S_4 => temp4,
        sorgente => source,
        destinazione => destination,
        D_1 => out0,
        D_2 => out1,
        D_3 => out2,
        D_4 => out3
    );
end Behavioral;

```

## switch\_elem\_tb.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity switch_elem_tb is
end switch_elem_tb;

architecture testbench of switch_elem_tb is
    signal X_1, X_2, Y_1, Y_2: std_logic_vector(1 downto 0);
    signal src, dest: std_logic := '0';

    constant CLOCK_PERIOD: time := 10 ns;

begin
    uut : entity work.switch_elem
        port map(
            X_1 => X_1,
            X_2 => X_2,
            src => src,
            dest => dest,
            Y_1 => Y_1,
            Y_2 => Y_2
        );

    process
    begin
        while now < 1000 ns loop
            wait for CLOCK_PERIOD / 2;
            wait for CLOCK_PERIOD / 2;
        end loop;
        wait;
    end process;

    process
    begin
        X_1 <= "00";
        X_2 <= "11";
        src <= '0';
        dest <= '0';
        wait for 20 ns;
        assert Y_1 = "00" report "Error for src = '0', dest = '0'" severity
error;
```

```

        assert Y_2 = "--" report "Error for src = '0', dest = '0'" severity
error;

X_1 <= "01";
X_2 <= "10";
src <= '1';
dest <= '1';
wait for 20 ns;
assert Y_1 = "--" report "Error for src = '1', dest = '1'" severity
error;
assert Y_2 = "10" report "Error for src = '1', dest = '1'" severity
error;

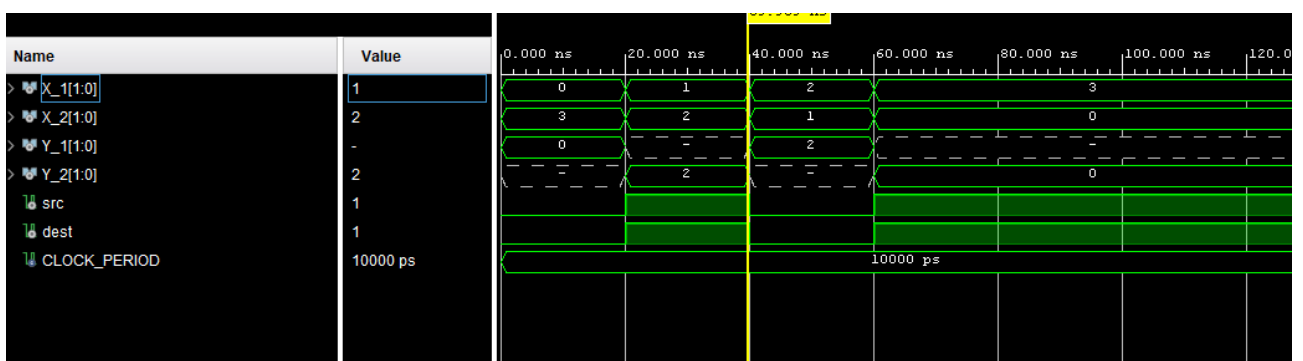
X_1 <= "10";
X_2 <= "01";
src <= '0';
dest <= '0';
wait for 20 ns;
assert Y_1 = "10" report "Error for src = '0', dest = '0'" severity
error;
assert Y_2 = "--" report "Error for src = '0', dest = '0'" severity
error;

X_1 <= "11";
X_2 <= "00";
src <= '1';
dest <= '1';
wait for 20 ns;
assert Y_1 = "--" report "Error for src = '1', dest = '1'" severity
error;
assert Y_2 = "00" report "Error for src = '1', dest = '1'" severity
error;

wait;
end process;

end testbench;

```





## rete\_di\_interconnessione\_tb.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity rete_di_interconnessione_tb is
end rete_di_interconnessione_tb;

architecture testbench of rete_di_interconnessione_tb is
    signal S_1, S_2, S_3, S_4, D_1, D_2, D_3, D_4, sorgente, destinazione:
std_logic_vector(1 downto 0);

    constant CLOCK_PERIOD: time := 10 ns;

begin
    dut : entity work.rete_di_interconnessione
        port map(
            S_1 => S_1,
            S_2 => S_2,
            S_3 => S_3,
            S_4 => S_4,
            D_1 => D_1,
            D_2 => D_2,
            D_3 => D_3,
            D_4 => D_4,
            sorgente => sorgente,
            destinazione => destinazione
        );

    process
    begin
        while now < 1000 ns loop
            wait for CLOCK_PERIOD / 2;
            wait for CLOCK_PERIOD / 2;
        end loop;
        wait;
    end process;

    process
    begin
        S_1 <= "00";
        S_2 <= "11";
        S_3 <= "10";
        S_4 <= "01";
        sorgente <= "01";
        destinazione <= "10";
        wait for 20 ns;

        assert D_1 = "10" report "Error for Test 1" severity error;
        assert D_2 = "11" report "Error for Test 1" severity error;
```

```

assert D_3 = "--" report "Error for Test 1" severity error;
assert D_4 = "--" report "Error for Test 1" severity error;

sorgente <= "10";
destinazione <= "01";
wait for 20 ns;

assert D_1 = "--" report "Error for Test 2" severity error;
assert D_2 = "--" report "Error for Test 2" severity error;
assert D_3 = "00" report "Error for Test 2" severity error;
assert D_4 = "11" report "Error for Test 2" severity error;

sorgente <= "00";
destinazione <= "11";
wait for 20 ns;

assert D_1 = "--" report "Error for Test 3" severity error;
assert D_2 = "--" report "Error for Test 3" severity error;
assert D_3 = "10" report "Error for Test 3" severity error;
assert D_4 = "01" report "Error for Test 3" severity error;

sorgente <= "01";
destinazione <= "01";
wait for 20 ns;

assert D_1 = "01" report "Error for Test 4" severity error;
assert D_2 = "--" report "Error for Test 4" severity error;
assert D_3 = "--" report "Error for Test 4" severity error;
assert D_4 = "--" report "Error for Test 4" severity error;

sorgente <= "10";
destinazione <= "10";
wait for 20 ns;

assert D_1 = "--" report "Error for Test 5" severity error;
assert D_2 = "10" report "Error for Test 5" severity error;
assert D_3 = "--" report "Error for Test 5" severity error;
assert D_4 = "--" report "Error for Test 5" severity error;

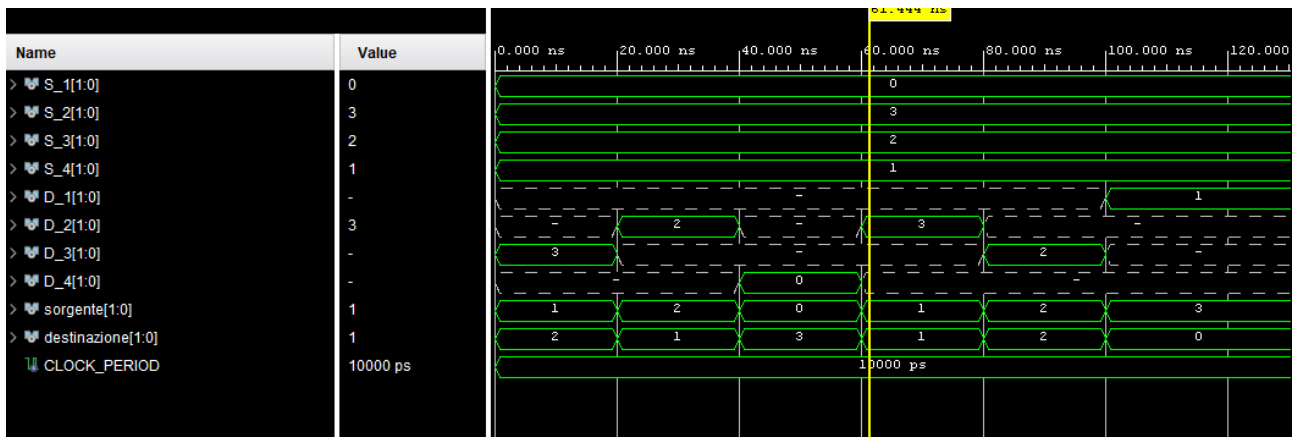
sorgente <= "11";
destinazione <= "00";
wait for 20 ns;

assert D_1 = "--" report "Error for Test 6" severity error;
assert D_2 = "--" report "Error for Test 6" severity error;
assert D_3 = "00" report "Error for Test 6" severity error;
assert D_4 = "--" report "Error for Test 6" severity error;

wait;
end process;

```

```
end testbench;
```



## tb\_sistema\_totale.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity tb_sistema_totale is
end tb_sistema_totale;

architecture tb_architecture of tb_sistema_totale is
    signal en0, en1, en2, en3 : std_logic;
    signal in0, in1, in2, in3 : std_logic_vector(5 downto 0);
    signal out0, out1, out2, out3 : std_logic_vector(1 downto 0);

    component sistema_totale
        Port (en0, en1, en2, en3 : in std_logic;
              in0, in1, in2, in3 : in std_logic_vector(5 downto 0);
              out0, out1, out2, out3 : out std_logic_vector(1 downto 0));
    end component;

begin
    uut: sistema_totale port map (
        en0 => en0,
        en1 => en1,
        en2 => en2,
        en3 => en3,
        in0 => in0,
        in1 => in1,
        in2 => in2,
        in3 => in3,
        out0 => out0,
```

```

    out1 => out1,
    out2 => out2,
    out3 => out3
);

stim_process: process
begin
    wait for 10 ns;
    en0 <= '1';
    en1 <= '1';
    en2 <= '1';
    en3 <= '1';

    in0 <="001100"; --11 a 2
    in1 <="011001"; --11 a 3
    in2 <="100110"; --10 a 0
    in3 <="110011"; --00 a 1

    wait for 60 ns;
    en0 <= '0';

    wait for 60ns;
    en1 <= '0';

    wait for 60ns;
    en2 <= '0';
    wait;
end process stim_process;

end tb_architecture;

```

